July 1983

The Model Concept: Nonprocedural Programming for Nonprogrammers

Stanley M. Schwartz
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-83-157.

The Model Concept: Nonprocedural Programming for Nonprogrammers

Abstract

This text is written as a self-contained guide to learning the MODEL language. MODEL is a radically new concept in nonprocedural programming, designed for use by individuals who do not have extensive computer experience. The text begins with a general introduction to computer concepts and data structures to provide background for the novice programmer. Later chapters cover a general overview of the MODEL language, MODEL language elements, MODEL data declaration, MODEL assertion statements, subscript use in assertions, and control variables. The text also includes an appendix of functions usuable in MODEL and an index. In keeping with the goal of being a self-instructional tool, there is a heavy emphasis on examples, and several complete MODEL specifications are explained in detail.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-83-157.

University of Pennsylvania
Department of Computer and Information Science
Moore School of Electrical Engineering
Philadelphia, Pennsylvania 19104

THE MODEL CONCEPT:

NONPROCEDURAL PROGRAMMING FOR NONPROGRAMMERS

by

Stanley M Schwartz

Submitted to
Information System Program
Office of Naval Research
Under Contract N00014-76-0-0416

Summer 1983

Moore School Report

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): The MODEL Concept: Nonprocedural Programming for Nonprogrammers	5. TYPE OF REPORT & PERIOD COVERED: Moore School Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Stanley M. Schwartz	8. CONTRACT OR GRANT NUMBER(s): N00014-76-0-0416
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Pennsylvania, Moore School of Electrical Engineering, Department of Computer Science, Philadelphia, PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Code 437, Arlington, VA 22217	12. REPORT DATE: Summer 1983
	13. NUMBER OF PAGES: 128 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

ABSTRACT

This text is written as a self-contained guide to learning the MODEL language. MODEL is a radically new concept in nonprocedural programming, designed for use by individuals who do not have extensive computer experience. The text begins with a general introduction to computer concepts and data structures to provide background for the novice programmer. Later chapters cover a general overview of the MODEL language, MODEL language elements, MODEL data declaration, MODEL assertion statements, subscript use in assertions, and control variables. The text also includes an appendix of functions usuable in MODEL and an index. In keeping with the goal of being a self-instructional tool, there is a heavy emphasis on examples, and several complete MODEL specifications are explained in detail.

ACKNOWLEDGEMENTS

Thanks to Dr. Noah Prywes and Dr. Jon Baron for their critical and incisive ideas without which this work would never have been written, to Yuan Shi for his tireless answering of questions, to the rest of the Project staff: Evan, Wu Hung, Adele, and Bolek, for their assistance in times of need, and to Lisa for her loving support.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1 HOW TO USE THIS TEXT

MODEL is an outgrowth of a recent movement in computer science to make programming less effortful. This evolution is taking place through the development of more sophisticated, higher level computer languages. The higher the level of a language, the less demanding and specific are the informational requirements that you, the user, must satisfy in order to sucessfully complete a programming task. In other words, higher level computer languages do more and more with less and less. MODEL is a very high level programming language. As you will see, by using MODEL, you can calculate the values satisfying a set of equations merely by entering the equations in any order and describing how the data to be used in the equations are organized. Because the amount of information you need to enter is so much less than in a conventional computer program, programs in MODEL are given a new name. Instead of programs, they are called specifications. When you "run" a MODEL specification, the programming drudgework of ordering the required calculations in proper sequence will automatically be done for you.

Because MODEL was intended to be easy to use, minimal assumptions are made about the computer sophistication of individuals who will be learning to use it. Therefore, this text is written for people who are familiar with some technical or business field where computers are used, but who have had no previous experience with computers. We will try to explain all the technical terms we use that are peculiar to computers, but we do not explain terms that you could have learned from the study of algebra or other basic mathematics. We also include comments written

for people who know more about computers, but these are usually in parentheses. In general, if you don't understand something in parentheses, don't worry. Learning MODEL will take some effort, if you have had no previous computer experience. Even if you have, there is much detail to learn if you want to master MODEL. However, with sufficient practice, you will become proficient in using MODEL, and then you will find the rewards in terms of programming ease and flexibility to be well worth it.

Our first aim here is to include all the information you will need on MODEL. This text may be used as a supplement to a programming course, but it is primarily intended as a self-sufficient guide with which the persistent reader can learn to use MODEL effectively. We imagine that most of you are learning MODEL for a specific purpose. Therefore, you may want to read certain sections fairly quickly if they do not apply to your application. We will try to explain everything in a way that is understandable, with a focus on examples. We will also include sample MODEL specifications, which should guide you in writing your own. The real test of your understanding will be your actual attempt to use the language. Hands-on experience writing and debugging your own MODEL specifications on a computer will be vital.

This text is laid out in chapters, each of which is divided into sections. The first chapter explains some of the philosophy behind MODEL and how MODEL is used. This chapter gives suggestions for using the rest of the text, and presents the sequence of steps in writing a specification. The second chapter provides a review of the basic computer concepts necessary to program in MODEL. Some of this material, especially the section about the organization of data structures, may be somewhat redundant for those of you who already have a computer background, but you should at least skim this material since many of these concepts are used in the MODEL language in a novel way. The third chapter provides an overview of the MODEL language, building from a single example. The fourth chapter lists the language elements that are common to every MODEL specification. The fifth chapter presents the syntax and semantics for describing the data to be used in writing MODEL

specifications. The sixth chapter describes in detail how equations are expressed in the MODEL language. This seventh chapter discusses subscripts, and the eighth describes the ways that MODEL control variables are used. Through the use of control variables in your equations, you can build flexibility into your specifications to allow them to handle different types of data in different ways.

We have several suggestions to make about the best way to read this text. You should first skim each chapter to get an idea of what it's about. Then you should read it with the idea of understanding - but not attempting to learn - what it says. You may want to make marks along the side of those passages you think you will need for a specific application. Plan to flip back to earlier sections to find information you need to understand later ones. The Table of Contents and Index, as well as cross-references in the text, should be useful for this. Do not get upset at your memory if you find you have to do a lot of rereading; there is much to learn, and it is simply not worthwhile to try to learn it all on the first pass. Try also to follow the explanations accompanying the examples. You may find it helpful to cover up explanations after you have just read them, and try to regenerate them yourself, as if you were teaching the example to someone else. This will help you to remember the important points, and also show you where there are gaps in your knowledge. In sum, although we do make an effort to include all the information you will need, even if you are a novice computer user, we can't lead you through it by the hand. You will have to do this for yourself.

1.2 USING THE MODEL SYSTEM

A computer program is a list of instructions that tell a computer what to do. It can be thought of as taking certain data (e.g., lists of numbers) as its input and producing other data as its output. The input can come from one or more of several external devices, such as magnetic tape, disk, cards, or a keyboard. The output can go to paper or a screen, as well as tape, disk, or cards. The MODEL system is itself a computer program, called a compiler, whose purpose is to produce another

program. It is thus an automatic program-writer. You, the user of the MODEL system, give the system a list of requirements, called a specification, as previously described, and the system automatically produces a program in PL/1, which you can then use as you would use any other program. (PL/1 is a high level, general purpose computer language, but it is not necessary for you to know how to program in PL/1 in order for you to use MODEL.)

The MODEL specification consists of three main parts: the Header names the specification and provides certain other information, the Data Declaration statements describe the organization of the data and the information to be manipulated, and the Assertion statements state the requirements to be met in the form of equations. Input data are called SOURCE data. Output data are called TARGET data.

The process of writing and using a MODEL specification takes place in several stages. These steps are illustrated in Figure 1.1. The process starts with your becoming aware of a data processing requirement, that is, a problem that you want to solve. In the second step you compose a MODEL specification in an attempt to solve the problem. Your MODEL specification can be composed with any external device that can be read into the MODEL compiler. Most users will probably enter their specification on disk, using a text editor at a keyboard.

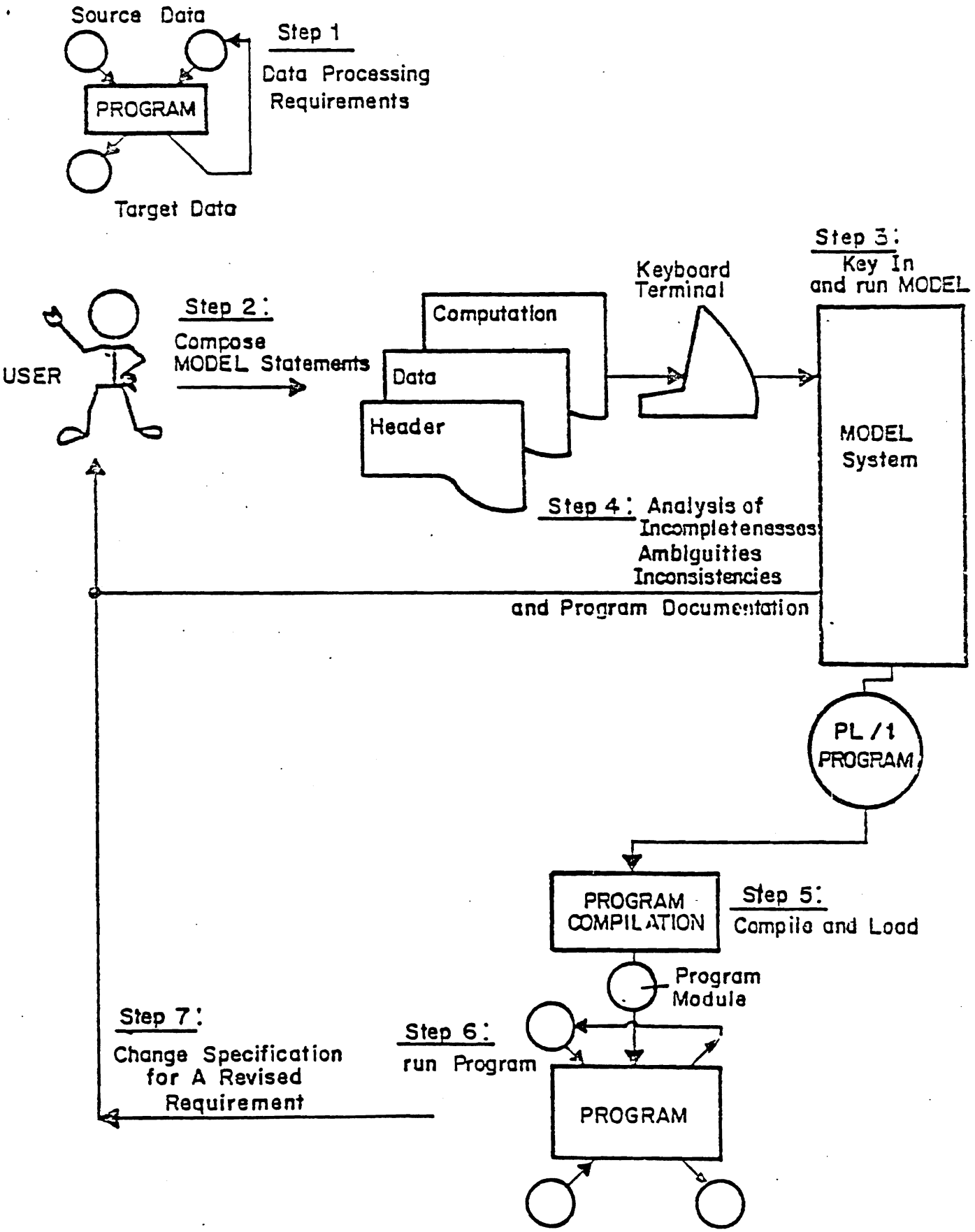


Figure 1.1 The Overall Procedure for Using MODEL

In step 3, the MODEL compiler checks for incompleteness, ambiguities and inconsistencies in your specification, and reports any that it finds. It tries to fill in what is missing in a sensible way. If it can do this, it produces the program in PL/1.

In step 4, it gives you documentation of its attempt to compile your specification. You receive a listing of the specification, a variable cross-reference report, a subscript-range report, a flowchart of the generated program, a listing of the generated program, and an error report. (You need not understand these terms now). The error report contains both warnings and error messages. Warnings indicate that in order to write the PL/1 program, aspects of your specification had to be reinterpreted. You may or may not agree with the reinterpretation, but in any case you will want to know about it, so that you can better understand the final output. Error messages refer to major problems that prevented the MODEL processor from successfully converting your specification into a workable PL/1 program. Each error message describes the type of error and the line in your specification where the error was uncovered.

At this point, using the documentation of the compiling of your specification, especially the error report, you may wish to correct and rewrite your specification. This process usually takes place over several stages. When you resubmit a specification, correcting the errors that the MODEL compiler first discovered, the processor may unearth other errors which were hidden. You will find this out in the error report on the resubmitted specification. For example, the processor cannot discover semantic errors in what a statement means, until errors in syntax, which make that statement uninterpretable in MODEL, are corrected.

In step 5, after you have successfully compiled your specification into a PL/1 program, you can enter that program into your system's PL/1 compiler and load it to be processed. Before you can run the PL/1 program, you must be sure that your SOURCE data are stored on some physical device, such as cards or disk, which is accessible to your system. The data may have already been stored on the device, or you may

have to enter them yourself.

In step 6, you run the program which the MODEL processor has written from your specification. It remains only for you to examine your TARGET data. If you are satisfied with the results, then you can stop. You can use this specification to solve similar problems in the future. Otherwise, in step 7, you can alter the specification and try again. You can also alter the specification to solve other processing problems. In general, it is easier to make alterations in the MODEL specification than in the PL/1 program it produces. For one thing, a specification is usually much shorter than the program it produces. For another, each variable is defined only once in the main part of a specification, although it may occur several times in the corresponding program, so the relevant information about it is easier to find. Thus, you should plan on making any needed changes in the specification rather than in the program. Because of the ease of making changes, MODEL is well suited to the maintainance of programs as well as their creation. You don't need to know anything about the program at all, except how to load it in and make it run, and what it is supposed to do, so that you can check to see that it is doing what it is supposed to.

1.3 WHY USE MODEL: A SUMMARY

MODEL is simpler to use than other languages, such as PL/1, because writing a MODEL specification depends merely on your being able to express the equations needed to calculate your results. Writing a MODEL specification is therefore much more conceptually allied to setting up equations for an algebra problem than to solving the equations or to writing a program in a conventional computer language such as PL/1. MODEL also helps you avoid many of the "nasty" parts of programming, such as input, output, program timing, memory allocation, and loops (repeated sets of operations), because these tasks are performed by the system. To use MODEL, you don't even need to know what a loop is. Therefore you can learn to use MODEL effectively even if you do not have years of exposure to computer concepts. Your task is further simplified because the MODEL system conducts a thorough analysis of your

specification and prompts you to correct any incompletenesses, inconsistencies, or ambiguities that it may have discovered.

MODEL will also be attractive to you if you are a proficient programmer with sophisticated business or scientific needs. Your assertion statements can take the form of simultaneous equations, making MODEL an ideal language in which to perform systems modelling. Unlike other very high level languages, MODEL is domain independent and general in purpose. MODEL permits all the functions and data types available in PL/1, and allows new functions to be defined as well. An additional advantage is that data description in MODEL is independent of the external device used for storage, simplifying input and output.

CHAPTER 2

BASIC CONCEPTS

2.1 MODEL TERMS

In this chapter, we will introduce some basic concepts of programming and data structures. You will need to read the following even if you are familiar with other languages, because in the MODEL language these concepts have a different meaning, even when the same term is used to refer to them.

2.1.1 FILES

Data are stored in FILES. A FILE is essentially a list of information, which may be stored on an external device. You can think of the purpose of a program as being the creation of an output FILE according to certain instructions, which may involve an input FILE. In MODEL, FILES containing input or SOURCE data are called SOURCE FILES. FILES containing output or TARGET data are called TARGET FILES.

2.1.2 VARIABLES

In a MODEL specification, requirements are expressed in terms of variables. (Usually, the same variables are also used in the PL/1 program that is produced, but, as we shall explain, the concept of a variable in MODEL is different from the usual concept in a program.) In most computer programs, a variable is a slot which is given the variable's name, like a person's mail slot. This slot can be filled with any one of a range of possible values, and these values can change over time. There are different kinds of variables, such as fixed decimal or character string. For an arithmetic variable, the slot can

contain a number; for a character string variable, it can contain a string of characters such as 'ABC' or 'FRED'.

In MODEL, a variable is like an unknown in an algebraic equation, which is defined uniquely according to its relationship to the other variables in the equation. Until a variable is used in an equation, it is undefined. A statement of a requirement in a MODEL specification is called an assertion statement. An assertion statement defines the value of a variable in terms of other variables. Once the value of a variable is defined, it cannot be changed. In this respect, MODEL is unlike most computer languages. The reason for this restriction in MODEL will be explained shortly. The only difference between MODEL variables and algebraic variables is that it is usual to use full names for the variables in MODEL, such as ITEM, instead of the single letters more common in algebra; with full names, it's easier to remember what the names mean.

2.1.3 SUBSCRIPTS

As in algebra, you can use a name with subscripts to distinguish several related variables. The subscript essentially gives a number to the variable as well as a name, so that different variables can be distinguished by number as well as name. Without subscripts, you would have to give different names to every number in a list. With subscripts, you use the same name, and different subscripts, for the different numbers in the list. Ordinarily, the subscripts start at 1 and go up to N, where N is the number of items in the list. A subscript is placed in parentheses after a variable name. For example, in ITEM(1) and ITEM(2), the subscripts are 1 and 2, respectively. These might be thought of as the first and second items, respectively, or as item number 1 and item number 2. Other variables can also be used as subscripts. For example, ITEM(I) can be used to refer to the Ith item. Here, I is a variable used as a subscript. If I has been set equal to 10, then ITEM(I) will be the same as ITEM(10). However, subscripts must be integers, or variables that have the value of integers; an expression such as ITEM(3.14) cannot be used. A variable may have more

than one subscript, and we shall explain at some length how such variables are handled.

2.1.4 ASSERTION STATEMENTS

The requirements in a specification are expressed in assertion statements. These are statements that define the value of one variable (possibly a subscripted variable) in terms of one or more other variables, constants, and functions. (This is different from other computer languages in which a temporary value for a variable is given to a slot using an assignment statement.) For example,

```
FRED = 5;
```

would define the value of the variable FRED as 5. FRED could then never be given another value in the same specification.

```
ITEM(2) = ITEM(1);
```

would define the value of ITEM(2) as equal to the value of ITEM(1). As we will explain presently, this assertion could appear at any point in the specification, even before the value of ITEM(1) was defined. The value of ITEM(1) would be defined in another assertion, such as

```
ITEM(1) = 6;
```

In writing assertions in MODEL specifications, it is more common (and much more efficient) to use variables as subscripts, instead of specific numbers. In this way, all the members of a long list could be defined at one time. Suppose we have a SOURCE FILE containing two variables called DIVIDEND(I) and DIVISOR(I), and a TARGET FILE containing a variable called QUOTIENT(I). Essentially, we have a list of dividends and a list of divisors, and we want a list of their respective quotients. The values of all the quotients in the TARGET FILE could be defined in a single assertion statement:

```
QUOTIENT( I ) = DIVIDEND( I ) / DIVISOR( I );
```

In words, this would mean, "The Ith quotient is defined as the Ith dividend divided by the Ith divisor." This would apply automatically and simultaneously for all values of I.

Often, it is inconvenient to define TARGET FILE variables directly in terms of SOURCE FILES variables, as in this example. Instead, we may have to make up other variables, that are defined in terms of SOURCE FILES variables, still other variables defined in terms of these, and so on. These variables are called intermediate variables. For example, QUOTIENT(I) might be an intermediate variable instead of the TARGET variable, and there might be another assertion statement defining some other variable in terms of QUOTIENT(I), such as

```
LOGQUO( I ) = LOG( QUOTIENT( I ));
```

2.1.5 NONPROCEDURALITY

A MODEL specification is nonprocedural. This means that you do not need to list the requirements in any particular order. Therefore, you may list your assertion statements in the order most convenient for you, or in an arbitrary order, without changing the meaning of the specification. In the program that MODEL produces, your instructions will be reordered so that your requirements can be satisfied with maximum efficiency.

For example, if your instructions are

```
ABE = BETH + CAIN;
CAIN = DAVID/EDNA;
EDNA = 1;
DAVID = 2;
BETH = 3;
```

MODEL will not be confused by the fact that the variables needed for the first instruction are not themselves assigned values until later. (You might try to calculate the value of ABE yourself, to see how MODEL might accomplish this task.) Nonprocedurality of assertions is one of the main features distinguishing MODEL specifications from conventional programs using ordered lists of instructions.

2.1.6 THE PROBLEM OF DEFINING SINGLE VALUES FOR VARIABLES

Nonprocedurality also explains some of the other features of MODEL. In particular, it explains why you cannot change the value of a variable once its value has been defined. To do this, you would have to specify which value was to be defined first, and when the definition was to be made relative to other definitions. This would amount to specifying the order of definitions, which you cannot do. Another feature is that, in MODEL, there is only a single defining statement for each variable, that is, an assertion statement in which that variable appears on the left. Again, if there were more than one, it would be necessary to say which was first, which was second, and so on. (It is possible to use more than one assertion statement to define a single variable if each applies when a different exclusive condition is met. In that case there would be no necessity to order them, for only one could apply at any one time.)

Nonprocedurality may at first create some problems, especially for a person who is used to conventional computer languages. On the whole, however, it is usually much easier, once you become accustomed to it. (Those with no computer experience will probably find it more natural than other languages from the outset.) One persistent problem concerns those cases in which we think of the whole point of the program or specification as involving the updating of a single variable. For example, suppose we wish to find M factorial (M!). (M factorial is the product of all the integers from 1 to M.) To find M! we need to perform a series of multiplications. In a a procedural languge we would keep increasing the value of a single variable, N, as each additional number is multplied in. In MODEL we compute M factorial using a subscripted

variable N(I) and a subscript I which assumes values corresponding to the numbers of all the elements of N(I) from 1 to M. Each successive element of N(I) corresponds to what would be a reassignment of the value of N in a procedural language, with the subscript I indicating which successive element we mean.

This can be expressed in a single MODEL assertion statement like

```
N(I) = IF I = 1 THEN 1 ELSE I * N(I-1);
```

(The symbol * indicates multiplication.) The assertion says that if the value of the subscript I is 1, then the value of N(I) is 1, otherwise, the value of N(I) is the value of the "last" value of N(I), namely N(I-1), times the value of the subscript I. In this case N(5) would have a value of 120, that is 5! (5 factorial). We assume here that I has some maximum value, M, which is the value whose factorial we are seeking. (If you are familiar with other languages, you might worry that such a trick will use up too much memory, by creating lists of variables when single ones would do. Don't worry. The MODEL system figures out by itself how to optimize the use of memory. You should not think about memory at all, but if you must, think of it as infinite in capacity.)

2.2 PRINCIPLES OF DATA ORGANIZATION

As part of writing a MODEL specification you need to describe how your input and output data structures are organized. When writing a specification you should start with an idea of exactly what you plan to put in (your SOURCE FILES) and exactly what you plan on getting out (your TARGET FILES). The equations you will write in your assertion statements represent a bridge between these two sets of structures. The form they will take depends on how the SOURCE and TARGET FILES are structured. Therefore, correct data structuring is very important. The rest of this chapter will describe a set of conventions for identifying the parts of data structures. Later chapters will explain in detail how

these conventions are applied to data declaration in MODEL by using specific MODEL keywords to label various parts of your data structures.

2.2.1 ARRAYS AND TREES

The phrase "how your data are organized" refers to the names and subscripts you use to refer to each piece of data. In MODEL, you should think about your data as consisting of lists of basic elements such as numbers, or lists of lists, or lists of lists of lists, etc. Each list has a name and a repetition count, that is, the number of elements it contains. The repetition count indicates the maximum value a subscript can take. The use of subscripts is consistent with organizing data in terms of tables, which are two-dimensional arrays (the use of dimensions and subscripts in describing arrays will be explained shortly). Each row in a table may be thought of as a list, and the whole table may therefore be thought of as a list of lists. Figure 2.1 shows an array of shopping lists for three different stores, with each list containing four items.

		ITEM #			
		1	2	3	4
	LIST OF STORES				
1	SUPERMARKET	apples	milk	bread	cheese
2	DRUG STORE	band-aids	cough drops	comb	soap
3	SCHOOL BOOKSTORE	MODEL text	calendar	pens	notebook

Figure 2.1
Shopping List Array

If you were to enter the shopping list array as data for a computer to use, for example to keep track of your budget, you would probably enter it in a single continuous string of items like:

```
apples,milk,bread,cheese,band-aids,cough drops,comb,soap,MODEL text,
calendar,pens,notebook
```

Somewhere in your budget program, you could then relay the information that this string has a regular, underlying structure; it consists of three lists of four items each. Because the array is regular, you wouldn't have to specifically describe how many items were on each list. You can do this because each list has the same number of items. In a regular array each row should have the same number of entries as every other row. The same thing is true for the columns.

Suppose you suddenly remembered that you're having your parents over for dinner and you want to serve them chicken. After checking your refrigerator, you realize that you need to add chicken to your supermarket shopping list. However if you add chicken to the string of data for your budget program by inserting it between cheese and band-aids, this will cause confusion. The computer will read your data to mean that chicken is bought at the drug store and soap at the bookstore. To keep track of a set of lists where you can potentially have a different number of items on each list, you need a different method of structuring data than a regular array. One alternative is to structure data in terms of trees.

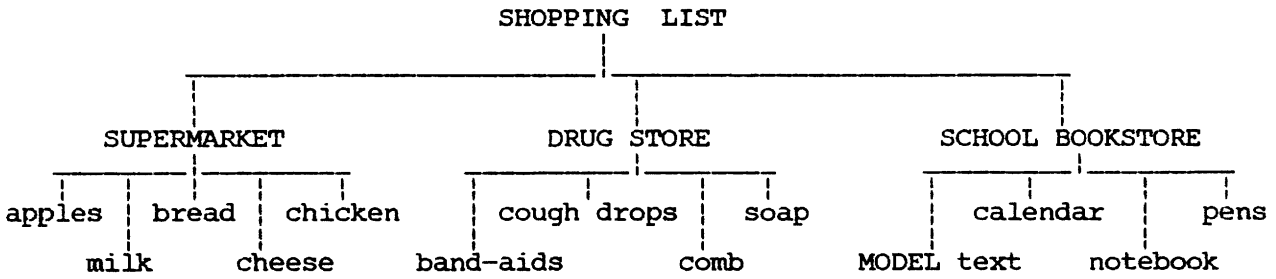


Figure 2.2
Shopping List Tree

The tree idea is more flexible because it allows different lists to contain different numbers of items. Figure 2.2 shows the shopping list array rearranged into a tree. Trees consist of nodes and the branches between them. Nodes are the parts of the tree that are individuated by names (or subscripts). In this case, the name of each store is a node in the tree. The individual items are also nodes. Pairs of nodes at

different levels in a data tree are connected by branches. These branches show how the nodes are interrelated. This kind of data organization is called hierarchical, because nodes representing lists that contains other lists are higher up in the tree than the nodes representing the lists they contain.

For any pair of nodes connected by a branch, the upper node is called the parent, and the lower node is called the child. One node can be the parent for any number of child nodes. Also the same node can be the parent of the one immediately below it in the tree and the child of the one immediately above it, just as a man can be the father of his daughter and the the son of his mother. (Although this form of data description is more flexible than arrays, it is also less efficient. As you will see, whenever you use trees for data description, it will be necessary to list how many child nodes each parent node is connected to.) The node SHOPPING LIST, representing the whole tree, is parent for the nodes representing shopping lists for individual stores. Similarly, store lists are parents for individual items.

2.2.2 DESCRIBING ARRAY STRUCTURE

Data description in arrays and trees in terms of how subscripts are used and dimensions are named is similar. Since more people are familiar with arrays (and more computer languages use arrays), the conventions of describing data in terms of arrays will be explained first. Then we can easily generalize these rules to apply them to trees. Data description in MODEL uses tree organization because of its greater flexibility. However for many applications, array description or tree description will be interchangeable.

The following descriptions of data structures use numbers as the basic elements. This is a matter of convenience. There is no rule prohibiting data structures from being constructed using character elements. There can be data structures of any size containing letters, names, binary bits, etc. There can also be data structures which include both numbers and character strings as basic elements.

A scalar or zero-dimensional array is a single variable, like X in algebra. See Figure 2.3(a) for examples. Giving a variable name, such as X or LUCY, to a scalar is enough to specify it uniquely. This means using the variable name, either you or the computer can refer to a scalar without getting it confused with any other scalar.

```
        .05              6.02E23              11
        LUCY             ETHEL          LITTLE_RICKY
```

(a) Examples of Scalars

```
        I          1      2      3      4      5
FELIX(I)         .04    .76    .31    .40    .72
```

(b) Example of a Vector

```
          ROW(I) 1        7   4   11   3
                 2        5   6    4   2
                 3        1   3    8  12

                          1   2    3   4
                             COLUMN(J)
```

(c) Example of Matrix MOE(I,J)

```
MATRIX(I)              1                       2

ROW(J)  1        7   4   1   3        8   9   9   1
        2        5   6   4   2        4   7   3   5
        3        1   3   8   2        2   6   5   6

                 1   2   3   4        1   2   3   4
                    COLUMN(K)               COLUMN(K)
```

(d) Example of Three-Dimensional Array CURLY(I,J,K)

Figure 2.3 Arrays of Different Sizes

A vector or one-dimensional array is a single row of variables, each of which can take on one value. See Figure 2.3(b) for an example of a vector. The number of elements (numbers) in a vector is called its range. Giving a variable name, such as FELIX, to a vector will not

enable you to specifically refer to any of its members. For a vector of range M, where M >= 1, any member of that vector can be uniquely specified with a subscript I, where 1 <= I <= M. From the example in Figure 2.3(b), FELIX(1) = .05, FELIX(2) = .76, FELIX(3) = .31, and so on. Each element of the vector has a different subscript. The whole vector is called FELIX(I), and its range is 5. Any data structure, such as FELIX(I), whose elements are distinguished with subscripts is called a repeating variable, or a subscripted variable.

An ordinary table of numbers is a two-dimensional array or matrix. A matrix consists of a certain number of vectors of the same length, laid down next to each other. Each vector is a row in the table, or if you prefer, each is a column. Each element of a matrix must have two subscripts, (I,J), to specify it uniquely, where one of the subscripts refers to the row number and the other refers to the column number. For example, in Figure 2.3(c), with I being row number and J being column number, MOE(2,3) = 4, while MOE(3,2) = 3. The range of the row dimension of MOE(I,J) is 3, while the range of the column dimension is 4.

A three-dimensional array consists of a series of matrices of the same shape arranged one above the other to form a rectangular prism, like a stack of blackboards. See Figure 2.3-d for an example (presented two-dimensionally). Each element of a three dimensional array must have three subscripts (I,J,K), where one of the subscripts refers to the number of the matrix, a second refers to row number, and a third refers to the column number. For example, in Figure 2.3-d, with I being matrix number, J being row number, and K being column number, CURLY(1,2,3) = 4, while CURLY(3,2,1) does not exist. The ranges of I, J, and K are 2, 3, and 4, respectively.

A four-dimensional array would consist of a series of rectangular prisms of the same size and shape, like several stacks of blackboards. The reader can probably infer that an element within it would require four subscripts (I,J,K,L) for unique specification. The general rule is that an array of n dimensions requires n subscripts in order for each element to be uniquely specified.

2.2.3 DESCRIBING TREE STRUCTURE

All data arrays can be restructured to form data trees, even if all data trees can't be restructured into data arrays. MODEL uses data description in terms of trees because of its greater flexibility. Figure 2.4 shows CURLY(I,J,K) laid out as a tree. The trunk or highest level node names the whole set of data. The trunk node then splits into major branches leading to the nodes on the next level. These nodes then split into subnodes, which ultimately split into the last level of nodes, the leaves of the tree.

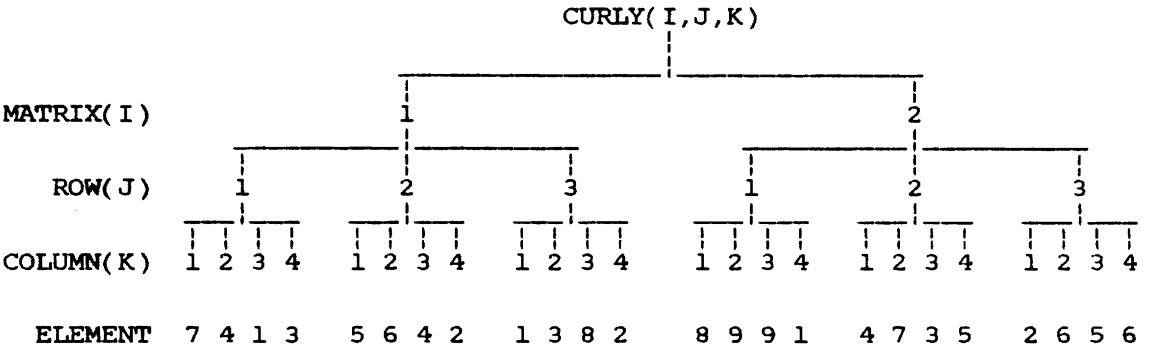


Figure 2.4

Tree Representation of Three-dimensional Array CURLY(I,J,K)

In MODEL, the highest level node naming the whole set of data is referred to as a FILE. The nodes in the middle of the tree are referred to as GROUPS or RECORDS; the bottom nodes are referred to as FIELDS. These leaf nodes represent the individual data points. Each time the tree divides, when going from the top of the tree to the bottom, another dimension is formed. The number of child nodes that the parent node immediately above that level of the tree divides into is the range of that dimension. In our example, the first division into two nodes corresponds to the vertical matrix dimension (I) of CURLY(I,J,K). (The range of this dimension is two, corresponding to the number of elements.) The second division into three subnodes for each node corresponds to the row dimension (J). The third division into four leaves for each subbranch corresponds to the column dimension (K). This example shows three dimensions, but the number can be as large or as

small as you desire.

In this kind of data hierarchy, the order of subscripts is the same as the order of the dimensions moving from the top to the bottom of the tree. The leftmost subscript corresponds to the division closest to the top of the tree (the matrix dimension in Figure 2.4). This is called the first dimension and represents the top level of the data hierarchy. Each subsequent division, moving farther away from the trunk of the tree, corresponds to the next dimension and the next subcript, moving left to right. The division farthest from the trunk of the tree (the column dimension in our example) corresponds to the rightmost subscript and last dimension. This is the lowest level of the data hierarchy.

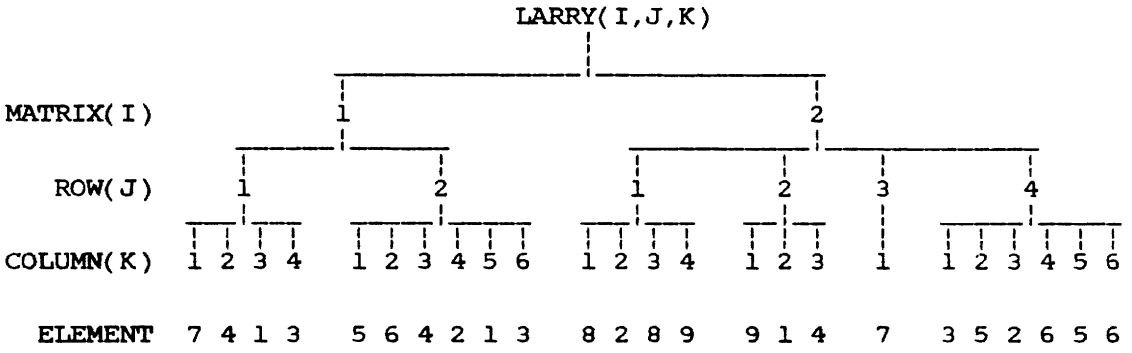
To refer to a particular piece of data, that is a particular FIELD, a specific value will have to be given to as many subscripts as there are dimensions in the tree. For the data-tree CURLY(I,J,K), three subscripts are required to specify each element. For example, the subscript (2,3,1) refers to the first (column) node branching from the third (row) node branching from the second (matrix) node. The value of this particular element is 2. The use of subscripts in assertions will be dealt with in Chapter 7.

One major advantage of organizing data hierarchically, is that this makes it easy for you to refer to elements of nonrectangular (jagged-edge) arrays. A nonrectangular array occurs when the data structures below a higher level dimension don't all have the same shape. An example would be a matrix made up of rows each containing a different number of elements, or a prism made up of matrices each containing a different number of rows. Figure 2.5(a) shows the elements of the array CURLY(I,J,K) arranged as a three-dimensional jagged-edged array, LARRY(I,J,K). This makes it difficult to keep track of individual elements. Figure 2.5(b) shows the same jagged-edged array rearranged as a hierarchical tree (LARRY(I,J,K)). The change from the previous tree structure using the same elements (CURLY(I,J,K), Figure 2.4), was easily implemented by increasing or decreasing the number of elements that the tree divided into at each level. For example, while both matrices in CURLY(I,J,K) have three rows, in LARRY(I,J,K) the first matrix (I = 1)

was given two rows, while the second (I = 2) was given four.

ROW(J)	1	7	4	1	3			8	2	8	9			
	2	5	6	4	2	1	3	9	1	4				
	3							7						
	4							3	5	2	6	5	6	
		1	2	3	4	5	6	1	2	3	4	5	6	
		COLUMN(K)						COLUMN(K)						

(a) Example of Jagged-Edged Three-Dimensional Array LARRY(I,J,K)



(b) Tree Representation of Jagged-Edged Array LARRY(I,J,K)

Figure 2.5

The other advantage of organizing data hierarchically is that this allows you to have trees which are not equally deep on both sides. This means that some of the nodes that the tree divides into at a particular level further subdivide, while others do not, (see Figure 2.6). The representation of data structures using trees is recursive. This is a computer science term which means that nodes can branch into new nodes which can further subdivide, and that the process can be continued indefinitely. This means that you have great freedom in designing trees to mirror the structure of your data, no matter how irregular it is.

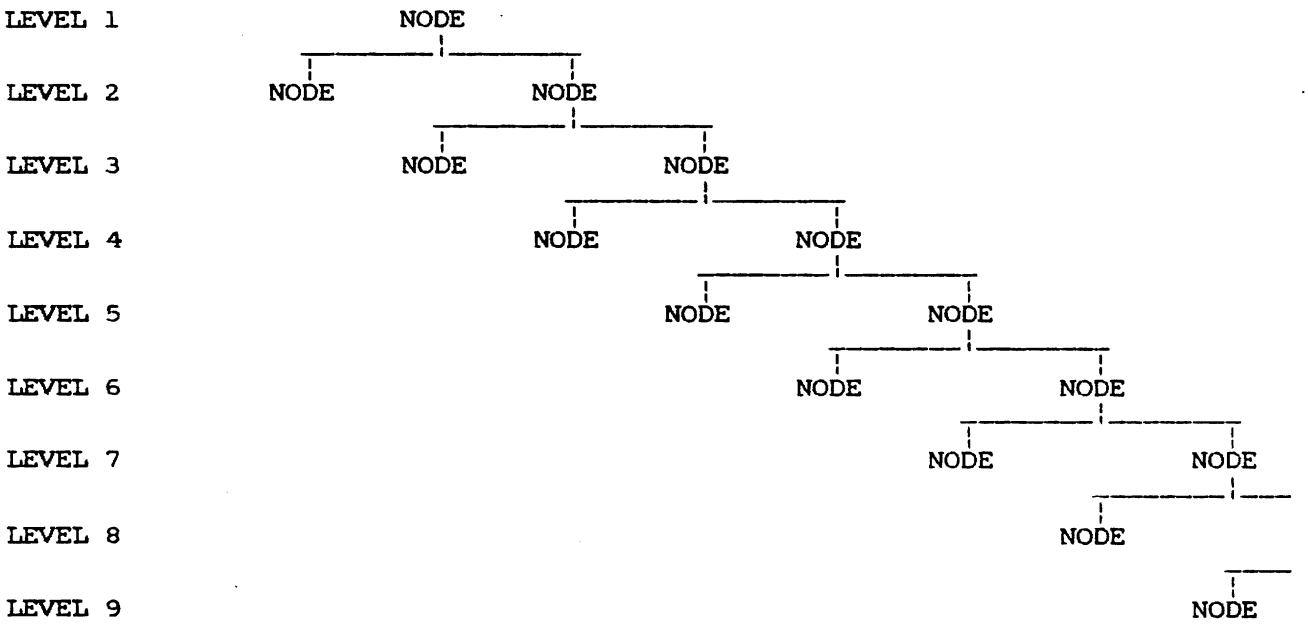


Figure 2.6

Illustration of Recursion in a Data Tree

CHAPTER 3

PARTS OF A SPECIFICATION

3.1 SAMPLE SPECIFICATION

The goal of this chapter is to give you an overview of the parts of a MODEL specification and how they are used together. The chapter starts off with a sample specification. This will give you an idea of how a MODEL specification is written. As we go further along in the text we will be referring back to this specification as a source of examples. The name of this specification is EXAMPLE. It is shown in Figure 3.1. The specification was written to solve a common problem found in evaluating progress in education. It could just as easily have been written to solve a similar problem encountered in business or in other domains.

When you look over the specification, you should focus mainly on learning how a MODEL specification is organized, that is, what things are necessary for you to include. To help you, as we describe each part of the specification we also cite the section of the text where that aspect of the MODEL language is described in detail. This should help you later when you want to write your own specifications. Following the specification, the rest of the chapter will continue with a more general and abstract discussion of the parts of a MODEL specification. Although this discussion will not be as detailed as in later, more specific, chapters, it should help you see how the different parts of the MODEL language tie together.

If you make mistakes in writing a specification, you can find out what they are, when you try to compile your MODEL specification into a PL/1 program. The MODEL system automatically sets up an error FILE, which you can read to find out what semantic and syntactic errors cropped up in your specification, and on what line they occured. (It's a good idea for you to get into the habit of checking the error FILE each time you try to compile a specification.) Errors which are not fatal to the PL/1 program, such as writing variable names which are too long, will appear as warnings in the error FILE.

Also as you read the text, you will notice that certain words are in all capital letters. Some of these are the names of variables. The other capitalized words such as MODULE, SOURCE, TARGET, FILE, RECORD, FIELD, SUB, ISAM, POINTER, KEY, GROUP, the names of functions, etc. are reserved words for the MODEL system. This means that these words have special predefined meanings for the system and that they cannot be used as variable names.

The problem concerns a class containing 12 students. During the semester, these students were given three tests. The problem consists of calculating the mean and standard deviation of the scores received by the students in the tests. Here, the mean is the same as an average, and the standard deviation is a common statistical measure consisting of the square root of the average of the squared differences between each score in a group and the mean of that group.

The specification starts off with a program header (discussed further in Section 5.2). The first line of the program header, the MODULE statement, names the specification, in this case EXAMPLE. The next statement gives the name of the SOURCE FILES. The SOURCE FILE is called SCORE. Its declaration consists of the layout of the data for the scores of each student on each test. The TARGET FILE statement in the program header contains the name of the TARGET FILE, STAT. Its declaration describes the layout of the data containing the means and standard deviations of individual test scores and total scores.

```
MODULE: EXAMPLE;
SOURCE: SCORE;
TARGET: STAT;

SCORE IS FILE (TEST_SCORE (3));
   TEST_SCORE IS RECORD (STUDENT (12));
      STUDENT IS FIELD (PIC 'Z99');

STAT IS FILE (TEST_STAT (3));
   TEST_STAT IS RECORD (MEAN_TEST,STD_TEST);
     (MEAN_TEST,STD_TEST) IS FIELD (PIC 'ZZZ9V.9999');

INT IS FILE (int_test (3));
   INT_TEST  IS GROUP (M_TEST (12),S_TEST (12));
      (M_TEST,S_TEST) IS FIELD (PIC '9999V.9999');

I IS SUBSCRIPT (3);
J IS SUBSCRIPT (12);

/*A1*/     M_TEST(I,J) = IF J > 1 THEN M_TEST(I,J-1) + STUDENT(I,J)
           ELSE STUDENT(I,J);

/*A2*/     MEAN_TEST(I) = M_TEST(I,12)/12;

/*A3*/     S_TEST(I,J) = IF J > 1
         THEN S_TEST(I,J-1) + ((STUDENT(I,J) - MEAN_TEST(I))**2)
         ELSE ((STUDENT(I,J) - MEAN_TEST(I))**2);

/*A4*/     STD_TEST(I) = (S_TEST(I,12)/12)**.5;
```

Figure 3.1

The Specification EXAMPLE

The layout of the SOURCE FILE is straightfoward, given the conceptualization of the problem. Since the data is sorted by tests, a natural first division is to divide SCORE into RECORDS by test. (FILE syntax is explained further in Section 5.4, while RECORD syntax is covered in Section 5.5.) (In MODEL, all data in SOURCE and TARGET FILES must be contained in RECORDS.) Therefore, in the SOURCE FILE we declare a RECORD variable called TEST_SCORE. Because there are three tests, this variable is given a repetition count of three. (Section 5.3 discusses the different ways allowed in MODEL to express repetition counts). Also since each RECORD contains the test scores of 12 students, the next natural division would be by student. We therefore declare a FIELD variable called STUDENT with a repetition count of 12 to denote the test scores of the students who took each of the three tests. (FIELD syntax is discussed in Section 5.6.)

The actual SOURCE data for the FILE SCORE is shown in Figure 3.2. (In this case the data was entered on disk using a text editor. Options for entering data in other ways are discussed in Section 5.4.) The data are laid out this way because of how we declared the data structure. Each RECORD in the SOURCE FILE correponds to one line of SOURCE data, that is, the scores of all the students on a particular test. Because there are three tests, the SOURCE data contains three lines. Each of these lines gives the test scores of 12 students. Identifying information, such as names or ID #'s, is not needed because the score for each student is distinguished by its position in the line. In this specification the data type of the FIELD variable STUDENT is declared to be (PIC 'Z99'). Data types are presented in detail in Section 5.6. This type specification means that each student's test score is read as having up to three digits. The Z means that a leading 0 on a two digit number may be entered as a blank, increasing clarity.

```
35 47 60 42 38 45 62 79 80 69 71 66
51 48 57 55 62 53 57 52 59 61 63 58
63 59 66 67 61 72 54 57 43 40 61 65
```

Figure 3.2

SOURCE Data SCORE

The layout of the STAT FILE is somewhat different. Since the means and standard deviations reflect the combined data of 12 students on each test, we no longer need a STUDENT FIELD variable. There is still a separate RECORD for each test, called TEST_STAT, but now it only contains one mean and one standard deviation FIELD variable for each of the three tests. The mean of the scores on each test is called MEAN_TEST and the standard deviation in called STD_TEST. The TARGET data produced from running the program produced by the MODEL system from the specification is shown in Figure 3.3. The labels were added afterward, but they could have been included as part of the specification. The means and standard deviations of scores on the three tests are expressed using a data type of (PIC 'ZZZ9V.9999'). This indicates that there are four possible digits to the left and to the

right of the decimal point, with leading 0's not printed.

	MEANS	SD'S
TEST 1	57.8333	15.2032
TEST 2	56.3333	4.4596
TEST 3	59.0000	9.0737

Figure 3.3

TARGET Data STAT

Data declaration in our specification also includes an interim FILE, INT. Interim variables (which do not need to be contained in FILES) are used when TARGET FILE variables cannot be easily defined directly in terms of SOURCE FILE variables, or as a convenience to allow you to use one variable to stand for a more complex expression that would appear in several assertion statements. (Interim data structures are discussed in Section 5.3). We will explain the interim data structure as we explain the purpose of each assertion.

We also declared two subscripts I and J. (Subscript declaration and usage is discussed in Section 6.4). Subscript variable I can take on integral values ranging from 1 to 3, and subscript variable J can take on integral values ranging from 1 to 12. Although MODEL allows optional subscript omission in certain situations, in the assertion statements of our sample specification we included the subscripts for the sake of clarity. You will notice that some variables, such as STUDENT, are always written using both an I and a J subscript. This is because we need to specify both the number of the student and the number of the test. Other variables, such as MEAN_TEST and STD_TEST, are always written with one subscript I. That's because means and standard deviations combine data across students, but not across tests.

Assertion statements in MODEL are written to define the values of TARGET FIELDS. In this specification, there are assertion statements to define means and to define standard deviations. (The syntax of assertion statements is discussed in detail in Chapter 6; the use of specific control variables in assertions is discussed in Chapter 8). In

procedural programs it is necessary to calculate means before standard deviations, because means are part of the formula for standard deviations. However, because MODEL is nonprocedural, the order of statements doesn't matter. We put the statements to define means first for ease of understanding.

A mean is equal to the sum of a series of numbers divided by the total number of members in the series. In our specification, defining a mean takes two assertions, one to sum scores together and another to divide the total by the number of members. It could have been done in one statement using the MODEL SUM function as in

```
MEAN_TEST( I ) = SUM( STUDENT( I,J )J )/12;
```

but we did it the longer way to provide a better example. (How functions are used is discussed in Section 4.10; a list of some of the functions available in MODEL is given as an Appendix.)

To find the sum of the scores of all the subjects taking each test, we defined an interim variable called M_TEST with the same repetition count as SUBJECT. (In this specification, all interim FIELDS have a data type of (PIC '9999V.9999'), which means four digits preceding and four digits following the decimal point.) We also defined an interim group called INT_TEST with the same repetition count as TEST_SCORE, so that we could keep track of the sums on all three tests.

In assertion A1, each element of M_TEST(I,J) is defined as equalling the sum of all the SUBJECT(I,J) scores (for test I) which have smaller or equal values of subscript variable J. (This is similar to the MODEL function RUN_SUM). The element of M_TEST(I,J) with the highest value of subscript J contains the total of all 12 SUBJECT(I,J) scores on test I. This element, divided by 12 in assertion A2, gives the mean of the scores of subjects taking the Ith test.

The standard deviation of a set of scores is equal to the square root of the average squared deviation (difference) of each score from the mean of that set of scores. Since standard deviations involve taking an average of the squared deviations from the mean, we first need to sum those deviations. To help with this we can create another interim variable with the the same repetition count as SUBJECT, called S_TEST. In assertion A3, each element of S_TEST(I,J) is defined as equalling the sum of all the squared deviations of SUBJECT(I,J) scores (for test I) from their test mean which have smaller or equal values of subscript variable J (relative to S_TEST(I,J)). The element of S_TEST(I,J) with the highest value of subscript J contains the total of the squared deviation scores for all 12 subjects taking test I. This element, which is divided by 12 and then raised to the .5 power in assertion A4, gives the standard deviation of the scores of all subjects taking the Ith test.

3.2 OVERVIEW OF THE PARTS OF A SPECIFICATION

The purpose of the rest of this chapter is to give an overview of the structure of a MODEL specification. As you saw in the sample specification EXAMPLE, a MODEL specification has three parts: program header, data declarations and assertions. The program header is used to identify the names of the specification, the name(s) of SOURCE FILE(S), and the name(s) of TARGET FILE(S) in three separate statements. You define the structure of your SOURCE and TARGET FILES, as well as any interim variables you need, in the data declaration section of your specification. Then, in the set of assertion statements, you define values for TARGET or interim variables. You can define an interim variable as equal to some complex expression and then use that variable in several assertion statements, instead of tediously copying the expression over and over.

Your SOURCE FILES are stored on external devices, for example on cards, tape, or disk. The TARGET FILES your specification produces will also be stored on external devices. However, any interim variables you create are stored only internally.

3.3 OVERVIEW OF DATA DECLARATION

Data declaration in MODEL is based on hierarchical tree organization, which allows great flexibility in data structuring, as was described in Section 2.2.3. We indicated the hierarchical structure of the sample specification EXAMPLE by listing the immediate descendents of a data node when that node (variable) was declared. MODEL data declaration also gives you the option of specifying tree structure through the order in which variables are declared as in

```
1 SCORE IS FILE,
   2 TEST_SCORE(3) IS RECORD,
      3 STUDENT(12) IS FIELD (PIC '99');
```

See Section 5.3.1 for further information on this alternate form.

You further define the structure of your data by giving ranges (repetition counts) for your variables. The MODEL language gives you several options for expressing ranges, for example

```
RACQUEL IS GROUP (SOPHIA (2));     means the range is 2.
RACQUEL IS GROUP (SOPHIA);         means the range is 1.
RACQUEL IS GROUP (SOPHIA (1:3)); means the range is variable, from 1 to 3.
RACQUEL IS GROUP (SOPHIA (*));     means the range is variable, maximum not
                                   known, but possibly very large.
```

You can also define ranges in assertion statements using control variables with SIZE or END qualifying prefixes as in

```
SIZE.SOPHIA = 2;  which means the range of Sophia is 2.
```

In the default case, when you do nothing to define variable ranges, they can sometimes be automatically deduced by the MODEL compiler from ENDFILE markers in the input or from the known ranges of accompanying subscripts. (ENDFILE markers are based on the physical characteristics of the data; you don't have to put them in.) See Sections 5.3.3, 7.3, 8.2, and 8.3 for more information on these options. In general, the

more specific you can be in defining variable ranges, the faster and more efficiently the compiler will run.

When declaring data for your specification, you should focus on how you want your data to be organized. What do your input data look like, and what would you like your output to be? For example, what are the most basic elements? Do certain lists or sets of elements have something in common so that you would prefer to refer to a group of them at one time, that is, make lists of lists? For example, can all the lists pertaining to a certain person be put together. Theses groups of lists, lists, and basic elements, correspond to MODEL FILES, GROUPS, RECORDS, and FIELDS, and form the different levels of your data structures.

All your SOURCE and TARGET data must be contained in FILES. Interim data may be stored in FILES, but you also have the option of declaring independent interim GROUPS or FIELDS. Each FILE declaration statement gives the name of the FILE. The FILE statement also lets the user specify that he wants the FILE to be keyed and organized index sequentially, as in

```
ALPHA IS FILE (BETA) KEY IS DELTA ORG IS ISAM;
```

This means that each RECORD BETA of the FILE ALPHA has a KEY FIELD (DELTA), which can be used in an assertion statement to refer to it, or look it up quickly, just as you can use a call number to locate a book in a library without looking thorugh all the books. This is faster than the normal method of locating RECORDS by their position in a sequence.

In the data declaration section of your specification you also give each GROUP or RECORD level variable a name and repetition count. A RECORD is chunk of data that is physically transferrable between an external FILE and the memory of the computer. All data structures, except for interim, must have their information contained in RECORDS. In other words, each SOURCE and TARGET FIELD must have one (and only one) RECORD above it in the data tree. GROUPS refer to any other intermediate sized data structures of a FILE which are not RECORDS.

GROUPS may appear in SOURCE, TARGET, or interim FILES, but are not mandatory in any of them.

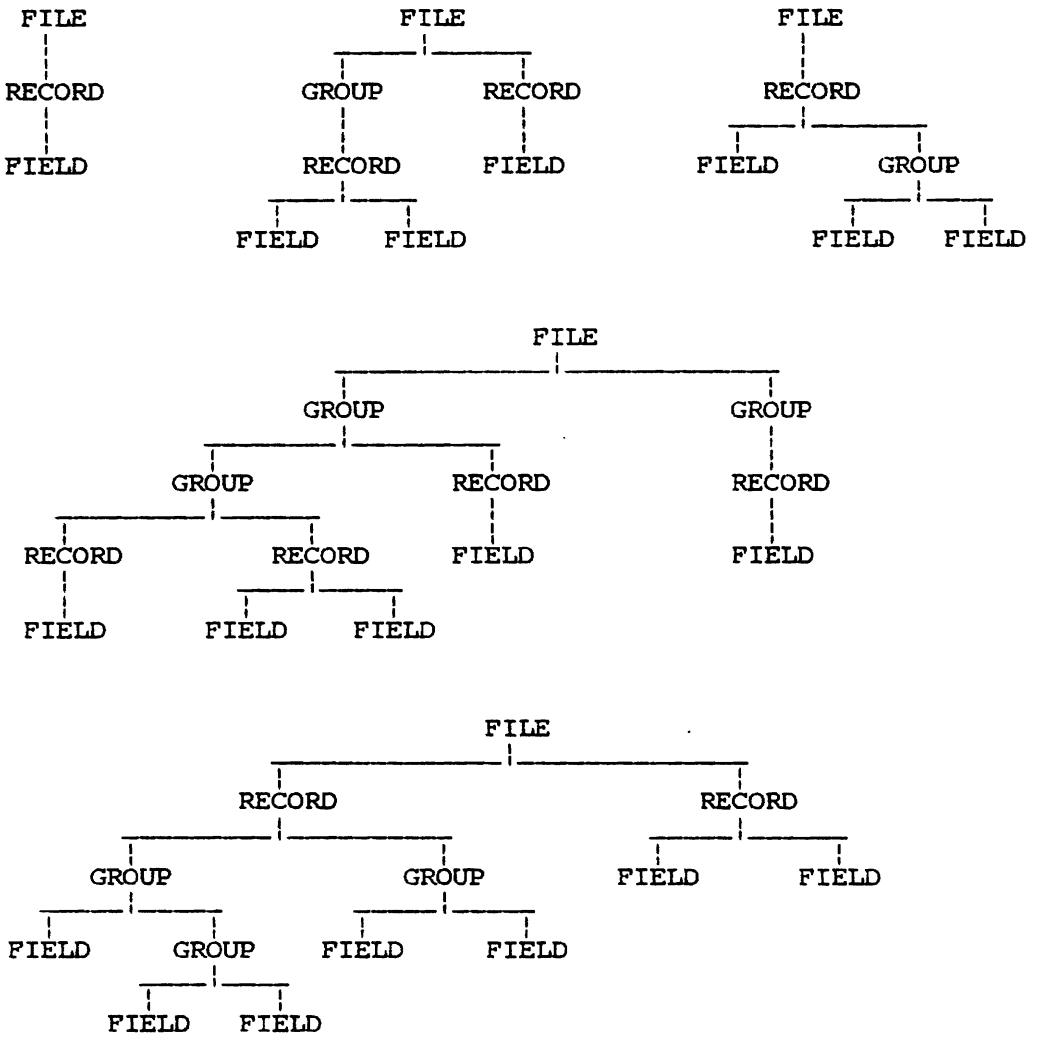


Figure 3.4

Some of the Many Legal FILE Structures Allowable in MODEL

Figure 3.4 shows that there can be GROUPS of RECORDS, GROUPS of GROUPS, and GROUPS of FIELDS (but not GROUPS of FILES). A RECORD could be made up of several GROUPS together or a mixture of GROUPS and FIELDS, but it cannot be made up of other RECORDS. Equally, a GROUP could consist of several RECORDS or a mixture of GROUPS (containing RECORDS) and RECORDS. However, a GROUP cannot have both RECORDS and FIELDS as

immediate descendents, because this would necessitate that some FIELDS have more or less than one RECORD above them in the data tree.

The declaration of FIELDS includes data type specification and expected length, as well as repetition counts. A FIELD is the smallest unit of a data tree, capable of holding a single piece of information (a name, a number, a word, etc). Each FIELD has a data type. Having a data type means that a FIELD can hold only certain kinds of information and be used in certain types of operations (see Sections 4.9 and 5.6). For example, arithmetic operations are defined for decimal, binary, or picture data types which contain numbers, but not for the character string data type which contains words.

Length is the number of characters per FIELD. You have several options for declaring the lengths of FIELDS, depending on the data type, including defining it in your assertion statements using the LEN-prefixed control variable as in

```
LEN.CARY = 5;
```

See Sections 5.6 and 8.4 for further information.

3.4 OVERVIEW OF MODEL ASSERTION STATEMENTS

You use assertions statements in your MODEL specification to define TARGET FILE or interim variables in terms of SOURCE FILE variables, functions, constants, and other TARGET or interim variables. Therefore, assertion statements represent links in a chain between SOURCE and TARGET data. Each assertion statement gives a separate equation to define each variable (although if two or more variables are equal, they may be defined in one equation). On the left hand side of the equation is placed the name of the variable to be defined. In mathematics, this would be called the dependent variable. On the right hand side of the equals sign is an arithmetic, logical, or character string expression composed of variables, constants, or functions whose value defines the dependent variable. In mathematics, these would be independent variables. (In a more complex conditional assertion statement, the

dependent variable may be defined in terms of any one of several expressions.)

All variables in assertion statements, both independent and dependent, are expressed as FIELDS. Each of these FIELDS, making up the lowest levels of SOURCE, TARGET, and interim data structures, can take only one value, as explained in Sections 2.1.5 and 2.1.6. This is in contrast to many computer languages in many values can be given to the same variable while the program is running. Consider as an example the statement X=X/2. In other languages, this tells the computer to divide the value in location X by two, creating a new value for location X. Such an interpretation would be inadmissible in MODEL. The reason is that you cannot specify the order in which assignments are made, because MODEL is nonprocedural. If you were allowed such statements as X=X/2, any other time you wrote X, e.g., Y=X+2, the system would have no way of knowing which value of X you meant, for in order to know this, it would have to know whether you intended the change in the value of X to be made before or after you used X to define Y. You should not think of your TARGET variables as being defined over time. The essence of MODEL nonproceduralness is all TARGET variables are conceived of as being defined simultaneously, so that the assertions form a set of true statements.

In MODEL, as in algebra, the equation X=X/2 does not mean "divide X by 2"; if such an equation were used at all, it would imply that X is 0! The MODEL equivalent of X=X/2 in the sense of changing X would be the statement:

```
X(I) = IF I = 1 THEN 100 ELSE X(I-1)/2;
```

This tells the computer to define the value of X in the Ith element of a variable X(I) equal to half the value of X in the I-1st element. For the first element of X(I) there can be no I-1st element. Therefore, we must define the value of that element separately. In this way, the assertion statement that we have written can be simulataneously applicable to all the elements making up the subscripted variable X(I).

3.4.1 TYPES OF ASSERTION STATEMENTS

Assertion statements in MODEL are of two types, simple and conditional. Simple assertion statements simply define the value of the dependent variable on the left side of the equation in terms of the expression on the right side (see Figure 3.5).

```
A = B + 3;
X(I) = Z(I) + 1;
Y(I,J) = J * 3**I;
```

Figure 3.5

Examples of Simple Assertion Statements

Conditional assertion statements define a dependent variable in terms of a particular expression when a certain condition is met. The condition is set off in the assertion statement by preceding it with an IF. Following that, the first (or only) expression which the dependent variable may be set equal to is preceded with a THEN. You can give the dependent variable another possible definition by including a second expression preceded by an ELSE, meaning otherwise. If the condition is met, then the dependent variable will be defined in terms of the first expression, otherwise it will be defined in terms of the second expression (or be undefined if no second expression is included.) MODEL was designed so that the value of each FIELD variable could be defined in a single assertion statement. However it is possible to use more than one assertion statement to define a single variable if each assertion statement applies only when a different exclusive condition is met.

It is also possible to nest additional conditions and defining expressions in conditional assertion statement. (Nesting in computer science means that one thing is included or embedded in another. In the case of conditions, nesting means that only if the first condition is met will the second be applied, and so on.) For example,

```
C = IF A = 0
    THEN IF B = 0
         THEN 1
         ELSE 2
    ELSE 3;
```

defines C as equalling 1 only when A and B both equal 0. When A equals 0 and B does not, then C is defined as equalling 2. When A doesn't equal 0, no matter what the value of B, then C is defined as equalling 3. (See Section 6.3.2 for more information on nesting in assertion statements.)

There are two ways of writing conditional assertion statements. (They are stylistically based on the computer languages PL/1 and ALGOL, respectively.) It doesn't matter which you use; the effect is the same. In one, you put the "IF" to the right of the "=" sign, that is,

```
variable = IF condition THEN expression1 ELSE expression2;
```

In the other, you put the condition on the left hand side of the "=" sign, that is,

```
IF condition THEN variable = expression1; ELSE variable = expression2;
```

Both of these forms are equivalent, in that both define the dependent variable in terms of the first expression if the condition is met, and in terms of the second expression if it is not. Figure 3.6 shows assertion A1 from our sample specification EXAMPLE written in terms of both types of syntax, so you can compare them. More examples, and a more formal presentation of the syntax of the two forms of conditional assertion statements is given in Chapter 6.

```
M_TEST(I,J) = IF J > 1
              THEN M_TEST(I,J-1) + STUDENT(I,J)
              ELSE STUDENT(I,J);

IF J > 1
THEN M_TEST(I,J) = M_TEST(I,J-1) + STUDENT(I,J);
ELSE M_TEST(I,J) = STUDENT(I,J);
```

Figure 3.6

The Two Forms of Conditional Assertion Statements

3.4.2 OVERVIEW OF SUBSCRIPT AND CONTROL VARIABLE USAGE

As stated previously, a subscript in a subscripted variable indicates the number of the particular element of the variable being refered to. A variable can take as a subscript any expression whose value is a positive integer ranging from one up to the the number of elements of a variable. For example, JOHN((1+2*4)/3) is a valid subscript for the variable JOHN, assuming JOHN contains at least three elements. Section 7.2 discusses the various types of subscript expressions, and Section 7.4 explains redundant subscript omission from assertion statements.

When writing assertion statements, you may sometimes wish to refer directly to data attributes, such as the number of repetitions of a repeating variable, the length of a piece of data, or the KEY FIELDS of an INDEX SEQUENTIAL FILE. To measure or define these attributes MODEL includes special control variables, which are formed by prefixing certain keywords to variable names to form qualified name variables. (How keyword prefixes are to be added is explained in Section 4.6). The control variables in MODEL for a data structure X include SIZE.X, END.X, LEN.X, NEXT.X, SUBSET.X, POINTER.X, FOUND.X, and MALDATA.X. The actions of all of the preceding control variables are described in detail in Chapter 8.

CHAPTER 4

COMMONLY USED LANGUAGE ELEMENTS

4.1 INTRODUCTION

This Chapter will tell you about the basic elements you can put together to write a MODEL specification. Here we will discuss the use of characters, operators, variable names, constants, expressions, and functions. We assume that you already know how to get access to the MODEL system. You can use a text editor to write your specification as an input file. (You also have the option of using other methods of input, such as tape or cards, but these are less convenient.) The individual statements you write are made up of the basic elements to be described in this Section.

4.2 EBNF NOTATION

In this Section we will explain EBNF notation, which is one commonly used means of expressing the syntax of statements in a computer language. (BNF stands for Backus-Nauer form, after its inventors. E stands for extended.) We use EBNF to construct syntax diagrams which give the allowable relationships between syntax elements. In these syntax diagrams, syntax elements consist of words, which are then combined to form clauses and expressions, which are then combined to form statements. Learning to read syntax diagrams written in EBNF is very important, because it's going to be our major way of outlining the parts of a MODEL specification and how they go together. We will use syntax diagrams in this Chapter to define the commonly used language elements and in subsequent Chapters to give the syntax of MODEL data declaration and assertion statements.

Syntax diagrams are expressed in EBNF using a set of symbols with special meanings. These symbols can be used to express how one syntax element is defined in terms of another, how parts of a statement are ordered, when elements are optional, and when they may be repeated. The following set of EBNF symbols will be used in syntax diagrams defining MODEL language elements:

1)	::=	means that the left hand side "is defined by" the right hand side.
2)	[...]	means that the enclosed is optional.
3)	¦	means that the immediate elements on both sides are alternatives, either of which can be used.
4)	[...]*	means that the enclosed repeats zero or more times.
5)	<...>	means that the enclosed will be defined in another statement in EBNF notation on the left hand side of ::= .

The following is an an example of a syntax diagram:

```
1  <MODEL specification> ::= [<statement>;]*
   2 <statement> ::= <header statement> ¦ <data declaration statement> ¦
                     <assertion statement>
```

The first line of the syntax diagram indicates that each MODEL specification is composed of any number of statements, each ending with a semi-colon. The second line indicates that a statement may be one of three types: a header statement (MODULE, SOURCE, and TARGET), a data declaration statement (FILE, GROUP, RECORD, and FIELD), or an assertion statement (simple and conditional).

4.3 MODEL CHARACTER SET

The character set used by the MODEL language consists of 82 characters, divided into classes of characters, digits, and delimiters, which are listed in Figure 4.1. You can use these characters to write words and to indicate operations. Words are either "reserved words," that is, words that have a meaning defined by the system (e.g., FILE, GROUP, IF, and the names of functions), or they are names of variables (e.g., TOTAL). Letters used in words in MODEL statements may be in either upper or lower case. However, for the purposes of this text,

reserved words in the text are noted by putting them in upper case letters. Characters, consisting of the letters, $, and _ can be used in MODEL variable names (see Section 4.5). The digits, the numbers from 0 to 9, can be used in variable names, except as the first character. They are also used in declaring FIELD data types. Delimiters, consisting of special characters, are used to separate words and expressions in MODEL statements. Other characters, which are not part of the MODEL language, such as @, ", ?, %, or # cannot be used in MODEL variable names or statements, but they can be used literally in character string constants or variables, as described in Sections 4.7 and 5.6.

```
<character> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|$|_|
                a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<delimiter> ::= .|<|(|+|&|||*|)|;|^|-|/|,|>|'|:|=|(BLANK)
<non-MODEL character> ::= @|"|?|%|#
```

Figure 4.1
MODEL Character Set

4.4 DELIMITERS

Delimiters, illustrated above, are used in MODEL to separate one word or one expression from another, that is, to show where one ends and the next begins.

Blank spaces are used to separate words, so that a blank space cannot be included in a variable name. However, extra spaces and blank lines are ignored, so you can format (arrange on the page) your specification any way you like.

In MODEL commas serve to separate elements in a series, such as variable names, subscripts, or function arguments (defined below).

All model statements end with a semi-colon ";". Usually, you write one statement per line, but a complex statement can run over several lines or several short statements can be put on a single line. A common syntax error is to omit the semicolon at the end of some of your statements. This error, while common, is difficult for the MODEL compiler to detect automatically, so it's important to remember to put a ";" at the end of each of your statements.

You can write comments, which are notes to yourself or others that have no effect on the system, by beginning your comment with "/*", slash-asterisk, and ending it with "*/", asterisk-slash as follows:

```
1  ‹comment› ::= /*‹text›*/
   2  ‹text› ::= [‹any character›]*
      3  ‹any character› ::= ‹character› ¦ ‹digit› ¦ ‹delimiter› ¦
                            ‹non-MODEL character›
```

Documenting your specification by writing comments to explain variable names or the purposes of assertion statements is a good technique, especially if other people will use a specification that you have written.

Parentheses are used to indicate that a list of elements or an expression is to be treated as a single element relative to the elements outside the parentheses (see Section 4.9.2 for more details and examples). Specific uses of parentheses in different types of MODEL statements are explained as each statement is discused in later chapters.

The use of most of the other delimiters will be explained in Section 4.8 on operators.

4.5 VARIABLE NAMES

Variable names must begin with a letter from the alphabet or the character "$" (dollar sign), but after that, they can include the numbers, and the character "_" (underline) as well. Therefore, the syntax diagram for variable names is as follows:

<name> ::= <character> [<character> | <digit>]*

Only the leftmost 8 characters of a variable name will be recognized by MODEL; the rest will be disregarded. Thus, the names PERCEPTION and PERCEPTIVE would both be treated as identical. You can make up the names of the variables as you please, but it's a good idea to make up names that are good mnemonics. For example, ITEM would be a good name for an element in a stock inventory.

The "_" is used to join parts of name, instead of a hyphen (which could be confused with a minus sign if it were allowed). If you want a variable name to consist of more than one word, than you should connect the words with "_", as in OUR_ITEM. You can still have a problem if you don't make the first 8 characters of each variable name, including "_", unique to each variable. For example, MODEL would treat OUR_FIRST_ITEM and OUR_FIRST_ WORD as the same variable, although it would give you a warning that it was shortening the names of variables with names longer than 8 characters.

4.6 QUALIFIED NAME VARIABLES AND RESERVED WORDS

The "." is used to join names to form a qualified name variable. This is a variable made from the name of a variable, preceding it with one or more special prefixes. The syntax of a qualified name variable is as follows:

```
1  <Qualified name variable> ::= <prefix>.[<prefix>.]*<name>
   2  <prefix> ::= <keyword prefix> | <parent name>
      3  <keyword prefix> ::= SIZE | END | LEN | NEXT | POINTER | FOUND |
                              SUBSET | FOR_EACH | MALDATA
      3  <parent name> ::= <name> | OLD | NEW
```

A qualified name may be much longer than 8 characters, including one or more prefixes. If two qualified name variables have the same prefixes, then each compound name is recognized through the first 8 characters of the name suffix.

A qualified name variable can be formed by giving a variable a keyword prefix, which is a "reserved word" in MODEL. (These keyword prefixes have a special meaning for the system and so cannot be used alone as variable names.) Qualified name variables with keyword prefixes are called control variables. They allow you to refer directly to some variable attribute, such as its range, in an assertion statement. For example, if ITEM is the name of a FIELD variable, then NEXT.ITEM is a qualified name variable that refers to the corresponding FIELD in the next RECORD. LEN.NEXT.ITEM is also a legal variable name, which shows that a single variable can have more than one keyword prefix. For a quick summary of the use of keyword prefixes see Figure 4.2.

END.X	denotes whether an element of a repeating variable X is the last one. Two value Boolean variable.
FOR_EACH.X	denotes global subscript with same range as the lowest, rightmost dimension of variable X.
FOUND.X	denotes if RECORD X exists in keyed FILE with same KEY as given by POINTER.X. Two value Boolean variable.
LEN.X	denotes the length of a FIELD variable X
NEXT.X	denotes FIELD X in the next sequentially ordered RECORD in a SOURCE FILE.
POINTER.X	denotes a KEY FIELD that references RECORD X in an ISAM FILE.
SIZE.X	denotes the range of the lowest, rightmost dimension of variable X.
SUBSET.X	denotes whether RECORD X is included or excluded in a subset of a FILE that forms a TARGET. Two value Boolean variable.
MALDATA.X	denotes whether a conversion error occurred when reading in a FIELD of SOURCE RECORD X. Two value Boolean variable.

FIGURE 4.2

Use of MODEL Keyword Prefixes

A second type of qualified name variable is used to eliminate ambiguity. This is done in two ways. One way is to give a variable used in an assertion statement a prefix of OLD or NEW. This differentiates between the same FIELD in a SOURCE FILE and its updated TARGET version (when both FILES and both FIELDS have the same name). An example is the assertion

```
NEW.BALANCE = OLD.BALANCE + DEPOSIT;
```

A second way is to give a variable a prefix of the name of the FILE where it is a memeber, if the same variable is used in two FILES with different names.

Consider the case wherein the same FIELD variable, ITEM, is contained in both a SOURCE FILE, IN, and a TARGET FILE, OUT. (IN and OUT are FILE names, not keywords.) For unambiguous reference in your assertion statements, the name of this FIELD could be preceded by the name of the FILE it came from as in

```
OUT.ITEM = IN.ITEM + SOMETHING;
```

4.7 CONSTANTS

A constant is a string of characters or numbers appearing in assertion statements. You write constants directly as operands in your assertions. This means that they are fixed, in the sense that their values don't depend on the data. MODEL recognizes three types of constants: character string constants, bit string constants, and arithmetic constants.

4.7.1 CHARACTER STRING CONSTANTS

Character string constants are defined as follows:

```
1  <character string constant> ::= '<any character> [<any character>]*'
   2  <any character> ::= <character> | <digit> | <delimiter> |
                          <non-MODEL character>
```

In other words a character string constant is formed by enclosing a string of any characters you choose of any length in apostrophes, for example, 'JON'. Character strings can be used with operators or functions that work on character strings, such as the concatenation operator, || (see next Section), or the function SUBSTRING (see Section 4.10 at the end of this Chapter).

4.7.2 BIT STRING CONSTANTS

A bit string constant is similar to a character string constant, but it can contain certain characters listed below. It is defined as follows:

```
1 <bit string constant> ::= '<bit> [<bit>]*'B[<n>]
  2 <bit> ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
  2 <n> ::= 1|2|3|4
```

Like a character string constant, a bit string constant can be of any length and should be enclosed with apostrophes. To distinguish it from a character string constant, you should also follow a bit string constant with a capital "B", as in '101011'B. MODEL accepts four forms of bit string constants. These are strings expressing values in base 2, base 4, base 8, or base 16. Bit string constants are used in logical and string expressions (see Sections 4.9.4 and 4.9.5). They cannot be used with arithmetic operators without conversion (see Section 4.11).

To show that a bit string constant is in base 2, you would follow the string with a B or B1. A base 2 bit string constant may only contain 1's and 0's. This is the form of bit strings which is most common. A base 4 bit string constant is indicated by following the string with a B2. It may contain the digits 0, 1, 2, and 3. Base 8 bit strings (octal) are indicated with a trailing B3 and may contain any of the digits from 0 to 7. Base 16 bit strings (hexadecimal) are indicated with a trailing B4. They can contain any of the digits from 0 to 9, and the letters A through F representing 10 through 15, respectively. For all the above forms of bit string constants, if you use digits or letters which are out of range of the specified base, you will get an error message.

4.7.3 ARITHMETIC CONSTANTS

Arithmetic constants are used in arithmetic operations and are written using a string of digits. They are defined as follows:

```
1   <arithmetic constant> ::= <unsigned number> [<exponent>]
    2  <unsigned number> ::= <unsigned integer> [<fraction>] |
                             <fraction>
       3  <unsigned integer> ::= <digit> [<digit>]*
       3  <fraction> ::= .<unsigned integer>
    2  <exponent> ::= E[<sign>] <digit> [<digit>]
       3  <sign> ::= + | -
```

Arithmetic constants are always decimal (not binary). They can have a fractional component (also decimal) and can be expressed in exponential notation. This is a shorthand way of expressing numbers which are very large or very close to 0. In exponential notation the unsigned number part of an arithmetic constant (called the mantissa) is multiplied by 10 raised to the power of the exponent. The exponent must be an integer. In the Vax version of Pl/1, the exponent cannot have an absolute value higher than 34 (see Section 5.6 for more information). The "E" preceding the number means "10 to the". Examples of arithmetic constants are 10011, 503.64, and 9.45E-23. (To write 9.45E-23 without exponential notation, you would have to put 22 zeros between the 9 and the decimal point at the beginning of the fraction.)

As you will see in the description of arithmetic expressions, arithmetic constants can also be given a positive or negative sign in front of them. Arithmetic expressions and the operators used in them will be discussed in detail in the next two Sections of this chapter.

4.8 OPERATORS

Certain of the MODEL delimiters, called operators, indicate arithmetic, logical, or string operations. These symbols are used with variables, called operands, to create expressions. There are

restrictions on which operator can go with which kind of operand (see Figure 4.3). These restrictions have to do with the idea of data types, which we shall discuss in detail in Section 5.6 of the next Chapter. For example, arithmetic operators can be used only with arithmetic constants or variables. It does not make sense to divide one character string by another character string. And although it may make sense to "add" two strings, that is, to string them end to end, MODEL uses "¦¦" for this rather than "+". This operation is actually called concatenation. Recall that when you declare variables in your specification, you also indicate the data type of each variable. If you then try to use the wrong operator with the wrong variable, for example, if you try to use a "+" between two variable names that stand for character strings, you should get a conversion error message (see Section 4.11).

The following symbols are the arithmetic operators which can be used with any arithmetic variables, constants, or functions that are decimal or binary. Arithmetic operators are used in arithmetic expressions (see next Section).

+ addition, or a prefix indicating a positive number;

- subtraction, or a negative number;

* multiplication - NOTE: This MUST be used for multiplication. Unlike algebra, you do not indicate multiplication by no symbol at all.

/ division

** exponentiation or raising to power - For example, 2**3 means 2 to the third power, or 8.

In addition to arithmetic operators, there are the following comparison operators, which are used in Boolean expressions (see next Section):

>	greater than
<	less than
=	equals
^>	not greater than
^<	not less than
>=	greater than or equal to
<=	less than or equal to
^=	not equal to

The equals sign ,"=", is also used in MODEL assertions to mean "is defined as".

Another set of operators, called logical operators, can be used in logical or Boolean expressions.

^	not
&	and
¦	or

These operators work on Boolean, or bit string variables (including binary). A Boolean variable has a value of either true or false represented by a 1 or a 0. In Boolean expressions, these operators work on single Boolean variables, while in logical expressions, these operators work on bit strings and binary variables (see Sections 4.7.2 and 4.9.4), which are equivalent to several Boolean variables strung together.

Parentheses, "(" and ")", are called group operators and can be used in any type of expression and with any data type. They ususally surround one expression which is contained within a second expression. They mean that the expression they surround is to be evaluated separately and treated as a unit in evaluating the second expression.

The symbol || is called the concatenation operator or string operator. It is used with bit strings and character strings in string expressions. It means "string together" or place end to end. Other operations performed on strings use reserved functions instead of operators, and we shall discuss these later.

4.9 EXPRESSIONS

Expressions are combinations of operators, functions, constants, and variables which act like clauses or phrases in a sentence to express a partial value. Expressions can also be combined to form larger expressions. The major types of expressions used in MODEL assertions are arithmetic, logical, string, and Boolean. Comparison expressions are a subclass subsumed under Boolean. The types of expressions, along with their preferred operators and operands are listed in Figure 4.3. It is also possible to write expressions containing mixed operands. Depending on the particular case, one of the operands may be converted by the PL/1 compiler, so that the expression can be evaluated, or you may get an error message. Section 4.11 describes the treatment of expressions with mixed operands by the MODEL system.

EXPRESSION	OPERATORS	OPERAND
ARITHMETIC	ARITHMETIC	ARITHMETIC (Decimal, Binary, Numeric String, Picture)
LOGICAL	LOGICAL	BIT STRING, BINARY
STRING	STRING (Concatenation)	STRING (Character, Bit)
BOOLEAN	LOGICAL	BIT, COMPARISON EXPRESSION
COMPARISON	COMPARISON	ANY EXPRESSION

Figure 4.3

Types of Expressions

4.9.1 USE OF OPERATORS IN EXPRESSIONS

Expressions containing operators are evaluated in the following order of priority, based on the operators they contain:

1. ( ) (expressions enclosed in parentheses)
2. functions
3. expressions used as subscripts (subscript expressions)
4. + (as a prefix, meaning a positive number)
   - (as a prefix, meaning a negative number)
   ** (exponentiation)
   ^ (logical not)
5. * (multiplication)
   / (division)
6. + (addition)
   - (subtraction)
7. || (concatenation)
8. = (equal to)
   > (greater than)
   < (less than)
   ^> (not greater than)
   ^< (not less than)
   ^= (not equal to)
   >= (greater than or equal to)
   <= (less than or equal to)
9. & (logical and)
10. | (logical or)

In this table, operators with higher priority (lower numbers in the table) are applied first in evaluating expressions. Thus, in A * B + C, A and B are multiplied before C is added. You can tell this because * is priority 5 and + is priority 7. Similarly in ^D || E, logical not, ^, is applied to bit string D before it is concatenated to bit string E, because ^ is priority 4 while || is priority 7. All the operators under the same number have the same priority. If several operators of the same priority are present in the same expression, then the operators will be applied sequentially from left to right, except for exponentiation which is applied from right to left.

4.9.2 PARENTHESES IN EXPRESSIONS

Expressions can be combined to form larger expressions. At the end of Chapter 2, we discussed structuring data in terms of trees where nodes can keep branching into new nodes deeper in the tree. The same thing is true for expressions; the various types of expressions can include expressions as basic elements. For example, the syntax diagram for arithmetic expressions (Section 4.9.3) shows that an arithmetic

expression, surrounded by parentheses, can take a sign, be raised to a power by an exponent, etc. These nested expressions could contain other expressions, which contained still other expressions, and so on.

You show that expressions are nested in other expressions by enclosing the inner expression in parentheses. By creating embedded expressions, you can change the order that operators will be applied in evaluating your expressions. This is because parentheses have the highest priority as operators. For example, in the expression A + B * C, according to the above order of operations, the multiplication is done before the addition. However if you rewrote the expression using parentheses to form (A + B) * C, the parentheses would cause the addition in the inner expression to be done first.

Prentheses can also be used to change the order that expressions containing logical and string operators are evaluated. For example, to apply a logical not, ^, to two strings after they have been concatenated, you should enclose the two strings and the concatenation operator in parentheses as in ^(A || B). If you use several sets of parentheses inside each other, the expression embedded within the innermost parentheses will be evaluated first, then the next innermost, and so on. You can also use parentheses to improve the readability of expressions by setting off related elements together, without changing the value of the expression, as in ((A**3)/B) - ((X**4)/Y).

4.9.3 ARITHMETIC EXPRESSIONS

The basic elements of arithmetic expressions are arithmetic constants, variables, functions, and other arithmetic expressions, as well as the various arithmetic operators (see Section 4.8). Any variables or functions used in arithmetic expressions, must have interpretable values in the sense that numbers can be substituted for them. Arithmetic variables may be of decimal, binary, numeric string, or picture data types, as described in the next chapter. Subscript variables are a special class of arithmetic variables, described in Chapter 7. (Arithmetic expressions used as subscripts, which may contain subscript variables, are called subscript expressions. These

are also described in Chapter 7.) Arithmetic constants are defined in Section 4.7.3.

The syntax diagram defining arithmetic expressions is as follows:

```
1  <arithmetic expression> ::= <arithmetic term> [<addition operator>
                               <arithmetic term>]*
   2  <addition operator> ::= + ¦ -
   2  <arithmetic term> ::= <arithmetic factor> [<multiplication operator>
                            <arithmetic factor>]*
      3  <multiplication operator> ::= * ¦ /
      3  <arithmetic factor> ::= <arithmetic primary>
                                 [** <arithmetic primary>]*
         4  <arithmetic primary> ::=   [<sign>] <arithmetic element>
         4  <sign> ::= + ¦ -
            5  <arithmetic element> ::= <arithmetic constant> ¦ <arithmetic
                                        variable> ¦ <subscript variable> ¦
                                        arithmetic function> ¦
                                        (<arithmetic expression>)
```

Operations in arithmetic expressions apply to both decimal and binary variables, functions, and nested expressions. (However, as described in Section 4.7.3, arithmetic constants can only be decimal.) It is also possible, to mix decimal and binary elements in the same expression.

The following are examples of arithmetic expressions:

```
4
ELEPHANT + 3
JACK * QUEEN
PAUL + DAVID * ANN/(LUKE - 4)
-12E-15
SIN(A**-2.75) + COSD(B**((4.79 +C) * 5.79 - D)
```

4.9.4 LOGICAL EXPRESSIONS

The syntax diagram of logical expressions is as follows:

```
1  <logical expression> ::=  <logical term> [¦ <logical term>]*
   2  <logical term> ::= <logical factor> [& <logical factor>]*
      3  <logical factor> ::= [^]<logical primary>
         4  <logical primary> ::= <bit string constant> ¦ <bit string
                                  variable> ¦ <binary variable> ¦
                                  <logical function> ¦ (<logical expression>)
```

The operands of logical expressions are bit strings (including functions, constants, and variables), binary variables, and nested expressions. (Bit string constants with a base other than 2 are represented internally as base 2 strings in logical expressions.) Each 1 or 0 in these operands corresponds to a separate Boolean variable. (In essence 1 means true and 0 means false.) The logical operators &, ¦, and ^ can be used with operands of any length; & and ¦ take two operands, while ^ takes one. The result of any of the above operations is always a single bit string.

The logical operators, and, &, and or, ¦, work on individual bits in the same respective positions from the right ends of two logical operands. When these operators are applied to operands of different lengths, 0's are added to the right of the shorter operand until the operands are of the same length. Thus, the length of the result will be the length of the longer operand.

If the logical operation is & and if the corresponding bits in the two operands are both 1's (true), then the resulting bit in the new string is a 1; otherwise it is a 0 (false). For the ¦ operation, if one or both of the bits in the two operands is a 1, then the resulting bit is also a 1; otherwise it is a 0. For example, if X is '10110'B and Y is '11000'B, then X & Y is '10000'B and X ¦ Y is '11110'B. '101111'B ¦ '1101'B & '11'B is '110000'B (remember that & is applied before ¦).

The logical not operator, ^, reverses the truth value of what it precedes; it exchanges '1'B's for '0'B's and '0'B's for '1'B's in a logical primary. For example, if X is '10110', then ^X is '01001'.

4.9.5 STRING EXPRESSIONS

String expressions, which can incorporate only the concatenation operator, ¦¦, are defined in the following syntax diagram:

```
1  <string expression> ::= <string term> [¦¦ <string term>]*
   2  <string term> ::= <character string constant> ¦ <character string
                        variable> ¦ <string function> ¦ <logical factor> ¦
                        (<string expression>)
```

In string expressions, character string constants, variables, functions, and nested expresions can be joined together to form longer character strings. For example, 'Learning the MODEL language' ¦¦ 'is fun.' becomes 'Learning the MODEL language is fun.' Logical factors can also be strung end to end using the concatenation operator, so that '111101'B ¦¦ ^'0101'B becomes '1111011010'B. Finally, you can concatenate character strings to logical factors. The end result will be a character string, with the logical factor being converted. For example, 'WILLIAM' ¦¦ '1001'B becomes 'WILLIAM1001'.

4.9.6 BOOLEAN EXPRESSIONS

Boolean expressions, like a Boolean variable, have a single value of true or false. Their syntax diagram is as follows:

```
1  <Boolean expression> ::= <Boolean term> [¦ <Boolean term>]*
   2  <Boolean term> ::= <Boolean factor> [& <Boolean factor>]*
      3  <Boolean factor> ::= [^] <comparison expression>
         4  <comparison expression> ::= <Boolean primary> [<comparison
                                        operator> <Boolean primary>]*
            5  <Boolean primary> ::= <arithemtic expression> ¦ <logical
                                     expression> ¦ <string expression> ¦
                                     (<Boolean expression>)
            5  <comparison operator> ::=  = ¦ < ¦ > ¦ <= ¦ >= ¦ ^= ¦
                                          ^< ¦ ^>
```

Boolean expressions can act like logical expressions and use the same operators. The only difference is that Boolean expressions can only have a single bit as an operand rather than bit strings of arbitrary length. Boolean expressions can use nested comparison expressions as operands, because comparison expressions have a single bit value. An example is the expression A = B ¦ C = D & E = F, which is true when either A equals B or both C equals D and E equals F.

4.9.7 COMPARISON EXPRESSIONS

Comparison expressions can use any type of expression as an operand. If the two operand expressions hold the relationship, such as equality, that is given by the comparison operator, then the comparison expression has a bit value of true. If not, then it has a bit value of false. For example, the comparison expression A > B has a value of true only when A is greater than B; otherwise it is false. When you write a comparison expression containing more than one non-nested comparison operator, such as A < B < 10, the comparisons will be made in order from left to right and judged true or false in turn. In this case the

comparison expression is always true, because the comparison of A and B will result in a true or false bit (no matter what A and B are), which will be converted to a 0 or a 1 for the next comparison which will be smaller than 10.

When two expressions are compared, they should be of the same type, that is, both arithmetic, both logical, both character string, or both nested Boolean. Also remember that comparison operators have a higher priority than logical and (&) and logical or (¦), so if you wish to compare two logical expressions containing & or ¦, you should enclose the logical expressions in parentheses.

When containing arithmetic constants, functions, and variables, comparison expressions follow the same principles as in arithmetic, for example, 3 >= 2 is true, while 7 < 5 is false. Comparison operations can involve bit or character strings as well as to numbers. When comparison operators are applied to bit strings, if the strings are of different length, then the shorter string is first extended with 0's on the right so that both strings will be of the same length. So, when evaluting the truth of '10010'B > '10001001'B, '10010'B will be extended to '10010000'B. (This is the same as the convention for using logical operators on strings of different length discussed earlier.) The bit strings are then compared as if they were binary numbers. Therefore, using the example from above, the Boolean expression '10010'B > '10001001'B is true, because '10010000'B is greater than '10001001'B. (If the numbers 10010 and 10001001 were directly compared in binary, then 10001001 would be greater.)

Character strings are compared character by character from left to right. If the character strings are of different lengths, blanks will be added to the right of the shorter string before the comparison is made. The comparison is alphanumeric, with letters near the end of the alphabet defined as greater than letters near the beginning of the alphabet, and lowercase letters greater than uppercase. Also letters are defined as greater than numbers, and numbers are defined as greater than most punctuation. (The actual basis for each comparison is the ASCII code numbers of the relevant characters.) If the first character

in two strings is the same, then the second is compared, and so on. All of the following are true comparisons of character strings:

```
 'ABZ' > 'ABY'
'NOAH' < 'Noah'
'catl' > 'cat.'
```

4.10 FUNCTIONS

There are certain tedious calculations that most users want their specifications to perform at one time or other. To save you from the necessity of having to write assertion statements to specify these calculations, you can use functions. These functions are either built into the MODEL system or added by a programmer. To use a function, you enter the name of the function along with the variable(s) or expression(s) that you want to be operated on. An example would be the function SQRT(X), which evaluates the square root of a positively valued arithmetic expression surrounded by parentheses. X is called the argument of the function; It's what the function operates on. In this case, the X represents an arithmetic argument, but for other types of functions, it could represent other types of arguments. More complex functions can have several arguments. The single result that a function with any number of arguments returns is called the function's value.

Function names in a MODEL specification are reserved words, which means they should not be used as variable names. MODEL can use any of the functions available in PL/1. Additional functions can also be written in PL/1 and added to your MODEL system. Once a function is created in this way, it's name also becomes a reserved word. A sample of PL/1 functions, plus a list of additional functions written especially for MODEL, is contained in the appendix.

The syntax diagram for functions is as follows:

```
1  <function> ::= <function name> [(<argument> [, <argument>]*)]
   2  <argument> ::=  <arithmetic expression> ¦ <logical expression> ¦
                      <character string expression>
```

The above diagram shows that to use a function you should type in the function's name followed by the variable(s) or expression(s) to be used as arguments, enclosed in one set of parentheses. If the function takes more than one argument, they should be separated by commas. Some functions do not take any arguments. In that case the parentheses should also be omitted.

As you can see in the appendix, there are functions appropriate for all expressions and data types. SQRT(X), which we already mentioned, is an ARITHMETIC function which can be applied to decimal, binary, numeric string, or picture data types. Another example of an ARITHMETIC function is the MODEL function SUM. SUM(X(I)I) adds up all the elements in vector X(I) and returns the result as a scalar. For a variable with two or more dimensions, a SUM function can be used to add the elements along any one of those dimensions. SUM is referred to as a reduction function, because it reduces the number of dimensions of the result to one less than originally contained in the argument.

Other functions, called string functions, can be used on character or bit string variables. An example would be the string function SUBSTR(X1,X2[,X3]). SUBSTR(X1,X2[,X3]) is used to extract a smaller string or substring from string variable X1. X2 gives an integral value corresponding to the position, counting characters from the left, where you want the substring to be extracted from X1 to start. You can use X3 to give the number of characters in the substring. X3 is optional, as indicated by the square brackets. If you omit it, then the substring will extend from position X2 to the end of X1.

4.11 CONVERSION OF DATA TYPES IN EXPRESSIONS AND FUNCTIONS

As explained in Section 4.9 and 4.10, MODEL expressions and functions were designed to use certain specific data types as operands and arguments. Figure 4.4 explains how the MODEL system evaluates expressions or functions written using nonpreferred data types.

TYPE OF LHS,RHS	OPERATOR
1,2,3,4,6,7	+,-,*,/,**
0,2,5,7	||
5	COMP*,&,|,^

LHS	OPERATOR	RHS
0	COMP*	0,2,7
2,7	COMP*	0,1,2,3,4,6,7
1,3,4,6	COMP*	1,2,3,4,6,7

* COMP= = | ^= | > | >= | < | <= | =(assignment)

0 CHARACTER STRING
1 FIXED BINARY
2 NUMERIC
3 FIXED DECIMAL
4 FLOAT BINARY
5 BIT STRING
6 FLOAT DECIMAL
7 PIC STRING

Figure 4.4

Conversion of Data Types in Expressions with Mixed Operands

For an operator in a given row of the above table, you can write a legal expression using as operands any data type mentioned in the LHS and RHS columns of the same row. When an assignment uses mixed operands, the RHS operand is automatically converted into the LHS type. Therefore, the evaluated expression will also be of LHS type.

In conditional assertion statements of the form:

```
A = IF C THEN EXP1
          ELSE EXP2;
```

C must have a bit string data type. The choice of data types for EXP1 and EXP2 obeys the above rule for comparison expressions. If different from A, the data types of EXP1 and EXP2 will be converted to be the same. (See Chapter 6 for more information on conditional assertion statements.)

Functions, as described in Section 4.10, have data types associated with their arguments and return values. When a function is wrtten using a nonpreferred data type, the rules of automatic convertion in the COMP row of the table are applied. This means that argument(s) will be converted to the type(s) conventionally used with the function, so that the function can be evaluated normally.

Warning messages are issued whenever automatic conversion is applied.

Error messages are issued whenever the above rules are violated.

CHAPTER 5

DATA DECLARATION IN MODEL

5.1 OVERVIEW

This Chapter will discuss the syntax and semantics of MODEL data declaration statements. Syntax, as you recall from the previous chapter, is concerned with how the elements of a statement are defined in terms of other elements and how they are ordered. Semantics, on the other hand, is concerned with the meanings produced by choosing in your statements to use certain elements rather than others or choosing to order them in certain ways. The discussion of the syntax of MODEL data declaration relies heavily on EBNF notation of syntax diagrams, which was introduced in Section 4.2. Therefore, you should make sure you understand how these diagrams are read before you go on. In the following sections these syntax rules will be applied in examples to give you some ideas about how best to use the variations in data declaration to solve particular problems. There is a lot of detail here, so the first time you read this Chapter, you should simply try to understand what you have read. Then, you can refer back to the appropriate Sections when you write specifications.

5.2 MODEL PROGRAM HEADER

MODEL specifications start with a program header. The header has three naming functions, it names the specification, SOURCE FILES, and TARGET FILES. These naming functions are carried out by using different statements as expressed in the following syntax diagram:

<header statement> ::= <MODULE statement> | <SOURCE FILE statement> |
<TARGET FILE statement>

The first statement is called the MODULE statement. Its syntax is as follows:

<MODULE statement> ::= MODULE: <name>;

An example of a MODULE statement would be: MODULE: FRED; In this example the word MODULE is a MODEL keyword, which means that it has a special predefined meaning. The keyword MODULE is used to name a specification. (This is different from the name of the file in which the specification is stored. The MODULE name is used internally to identify the PL/1 program compiled from your specification and the machine language code compiled from that program.) The example tells MODEL that FRED is the name of the specification. Each specification can have only one name. It is important to note that this statement, as do all MODEL statements, ends with a semi-colon.

The SOURCE FILE statement gives the names of SOURCE FILES. These serve as input FILES containing the data information to be used in the specification. They are presumably read in from external devices. The SOURCE FILE statement has the following syntax:

<SOURCE FILE statement> ::= SOURCE [FILE|FILES]: <name> [, <name>]*;

Examples of legal SOURCE FILE statements would be:

SOURCE: RALPH;
SOURCE: RALPH, NORTON;
SOURCE FILE: NORTON;
SOURCE FILES: NORTON, RALPH;

These examples show that in SOURCE FILE statements the words FILE or FILES are optional and if there is more than one SOURCE FILE, then they should be separated by commas. It is also possible to create a MODEL specification without SOURCE FILES, if you define all the dependent variables in your assertion statements in terms of constants, or variables that already have been defined in terms of constants. For example, the specification in Figure 5.1, which doesn't include a SOURCE FILE, will produce a series of 12 numbers, in which each number is equal to the sum of the two previous numbers in the series. (In mathematics this is known as the Fibonacci series.) It was necessary to define only the first two members of the series using constants; this provided enough information to define the others.

```
MODULE: SUM;
TARGET: OUTPUT;

1  OUTPUT IS FILE,
   2  RECOU (12) IS REC,
      3  E IS FLD (PIC 'ZZ9');

I IS SUBSCRIPT;

E(I) = IF I = 1 THEN 1 ELSE
       IF I = 2 THEN 1 ELSE E(I-1) + E(I-2);
```

Figure 5.1

Example of a Specification Without a SOURCE FILE

TARGET FILES are the output from running the program produced from your specification, presumably to be stored on some external device. They retain the results of the computation for examination or future use. The syntax of TARGET FILE statements is the same as for SOURCE FILE statements.

It is as follows:

‹TARGET FILE statement› ::= TARGET [FILE|FILES]: ‹name› [, ‹name›]*;

Examples of legal TARGET FILE statements would be:

```
TARGET: ALICE;
TARGET: ALICE, TRIXI;
TARGET FILE: TRIXI;
TARGET FILES: TRIXI, ALICE;
```

It is also possible to use the same FILE as both SOURCE and TARGET. To do this, you would include the same FILE name in both the SOURCE FILE statement and TARGET FILE statement. You then only have to declare the structure of the FILE once. Later, you should clarify to which FILE a particular variable in your assertions belongs by adding the qualifying prefixes OLD (for SOURCE) or NEW (for TARGET), as was described in Section 4.6. For example, Figure 5.6 in Section 5.4 shows the ISAM FILE GINGER used as both a SOURCE and TARGET FILE. This example also illustrates the use of the POINTER control variable, described in detail in Chapter 8.

5.3 DATA DECLARATION SYNTAX AND SEMANTICS

Since MODEL is a nonprocedural programming language, each variable can take only one value. Therefore each variable in MODEL is named to distinguish it from every other variable, and the elements of a repeating variable are given subscripts to distinguish them. Data declaration in MODEL allows you to express the hierarchical organization of your variables, whether external (SOURCE or TARGET FILES) or internal (INTERIM FILES, GROUPS, RECORDS, or FIELDS). Data declaration statements express several pieces of information. They tell what the name of each variable is and whether it refers to a large set of data, such as a FILE, a medium-sized set of data, such as a GROUP or RECORD, or a single unit of data, such as a FIELD. Two variables in the same FILE should not be given the same name. Data declaration statements

also denote the relative positions of variables in the data tree, that is, which GROUP is above which RECORD. For variables below the level of FILE, data declaration statements may give repetition counts, with the option of leaving them unspecified. For FILES they give optional information about external devices, and for FIELDS they give data types and other data attributes.

5.3.1 TWO TYPES OF DATA DECLARATION SYNTAX

MODEL has two different ways of expressing the interrelationships between variables in the data tree. Figure 5.2 (a and b) expresses the data declaration of the specification EXAMPLE, which was introduced in chapter 3, in terms of both types of syntax, so that you can compare them. In most respects, the first and second forms of MODEL data declaration syntax are used identically. In either, you have the same options for describing FILES (SAM versus ISAM), storing data on external devices, specifying repetition count information, declaring several variables together, or choosing FIELD data types. These aspects of data declaration in MODEL will be described later in this chapter. The differences between the first and second forms of MODEL data declaration will be described below.

```
SCORE IS FILE (TEST_SCORE (3));
   TEST_SCORE IS RECORD (STUDENT (12));
      STUDENT IS FIELD (PIC '99');

STAT IS FILE (TEST_STAT (3));
   TEST_STAT IS RECORD (MEAN_TEST, STD_TEST);
     (MEAN_TEST,STD_TEST) IS FIELD (PIC 'ZZZ9V.9999');

INT IS FILE (INT_TEST (3));
   INT_TEST  IS GROUP (M_TEST (12), S_TEST (12));
      (M_TEST,S_TEST) IS FIELD (PIC '9999V.9999');
```

(a) The Specification EXAMPLE Declared with the First Syntax

```
1 SCORE IS FILE,
   2 TEST_SCORE(3) IS RECORD,
      3 STUDENT(12) IS FIELD (PIC '99');

1 STAT IS FILE,
   2 TEST_STAT(3) IS RECORD,
     3 (MEANTEST,STDTEST) IS FIELD (PIC 'ZZZ9V.9999');

1 INT IS FILE,
   2 INT_TEST(3) IS GROUP,
      3 (M_TEST,S_TEST)(12) IS FIELD (PIC '9999V.9999');
```

(b) The Specification EXAMPLE Declared with the Second Syntax

Figure 5.2

Alternate Forms of Data Declaration for the Specification Example

The first type of syntax allows more freedom in the order of declaration of variables. In the first type of syntax (shown in Figure 5.3), there is one data declaration statement for each variable. The immediate descendents of a variable at a particular level of the data tree are also identified, with repetition counts, in the statement declaring their parent variable. This does not apply to the declaration statements of FIELD variables, since FIELDS have no descendents in the data tree. Otherwise the declaration statements for variables at different levels of the data tree are similar. (Because the semi-colons act as delimiters, if you wanted to, you could place the data declaration statements for several variables on the same line.)

```
1  <data declaration statement> ::= <variable> [<IS>] <variable level>
                                    <variable argument>;
   2  <variable> ::= (<name> [, <name>]*) | <name>
      3  <name> ::= <character> [<character> | <digit>]*
   2  <IS> ::= IS | ARE | =
   2  <variable-level> ::= <FILE> | <GROUP> | <RECORD> | <FIELD>
   2  <variable-arguments> ::= <FILE argument> | <descendents> |
                               [(]<data type definition>[)]
      3  <descendents> ::= (<name> [(<repetition count>)] [, <name>
                           [(<repetition count>)]]*)
         4  <repetition count> ::=  * | <number occurrance> |
                                    <min occurrance> : <max occurrance>
            5  <number occurrance> ::= <unsigned integer>
            5  <min occurrance> ::= <unsigned integer>
            5  <max occurrance> ::= <unsigned integer>
```

Figure 5.3

Syntax of the First Type of Data Declaration Statement

Double declaration helps to make the relationships between variables clearer than in the second type of syntax and allows data declaration statements to be presented in an arbitrary order. This is consistent with the nonprocedural philosphy of the MODEL language. Therefore the remainder of this Chapter, after this section, will focus on the use of the first type of syntax. The cost of this kind of syntax is that it requires variable names below the level of FILE to be typed in twice, once in their own declaration, and once in the declaration of their parent in the data tree.

The second type of syntax, shown in Figure 5.4, is less flexible but more terse. (It resembles data description in PL/1 or COBOL.) In this type of syntax, the structure of the whole data tree is described in a single statement ending with a semi-colon. Each variable is declared only once within the statement, with the declaration delimited by commas. The repetition count of each variable is therefore given as part of the declaration of the variable, rather than as part of the declaration of its parent.

```
1  <data declaration statement> ::= <level number> <variable> [(<repetition
                                    count>)] [<IS>] <variable level>
                                    [<FILE argument>]
                                    [ [( )]<data type definition>[ )] ]
                                    [, <level number> <variable>
                                    [(<repetition countt>)] [<IS>]
                                    <variable level>
                                    [ [( )]<data type definition>[ )] ]]*;
   2  <level number> ::= <unsigned integer>
```

Figure 5.4

Syntax of the Second Type of Data Declaration Statement

Variables are ordered depth first, left to right. This means that each variable to be declared is positioned in the statement so that it is before the variables it is above in the data tree and after the variables it is below. In general, the closer a variable is to the beginning of the statement, the closer it is to the top of the data tree, and the closer to the end, the closer to the bottom. To clear up ambiguities about whether several variables are immediately below the same parent or whether a variable is at the bottom of one branch or at the top of the next, the declaration of each variable starts with a level number which describes the depth at which the variable occurs in the tree. The top name in a data tree, which is a FILE, unless the tree is an interim data structure, must be given a level number of 1. It must also have a repetition count of 1. Several variables can have the same level number if they have the same parent variables, or if they had different parents at the same depth into the tree (see Figure 5.2 (b)).

MODEL allows you to have large or small increments (in integers) between successive levels of the data tree. It also allows you to use different level numbers for variables at the same depth into the tree, as long as these numbers are not higher than any of the numbers used on the next level down or lower than any used on the next level up. This means that the MODEL system is tolerant of user errors. However using non-consecutive numbering for your variables is not recommended, because it will make it more difficult for you to keep track of them.

You may use both syntaxes of data declaration in the description of a single FILE as long as you switch from the first form of syntax to the second, as in

```
RICK IS FILE (JASMINE);
1  JASMINE IS RECORD,
   2  CAROL (2) IS FIELD (PIC '99V.99');
```

This form of mixing syntax is limiting because the first variable declared in the second syntax must have a repetition count of 1. It also must be given a 1 as it's level number. Switching in the opposite direction in the declaration of a single FILE is not allowed. However, there is no problem in declaring different FILES in the same specification using different forms of syntax.

Some additional pieces of information about either syntax of MODEL data declaration will give you additional flexiblity in describing your data organization. If a variable has a repetition count of 1, then a repetition count number does not have to be included for that variable. You also have the option of leaving the range of a subscripted variable unspecified at the time of data declaration, as will be described in Section 5.3.3. Finally, initial indenting of lines, as shown in Figure 5.2, is not necessary, but can probably help you keep better track of the relationships between your variables.

5.3.2 SHORTCUTS IN DECLARATION OF DATA STRUCTURES

If you want to include the same data structure in two or more separate FILES, then data declaration in MODEL allows you to declare that redundant data structure only once. For example, you can declare two or more FILES in one statement, as in:

```
(MICKEY, PLUTO) ARE FILES (DONALD (3), DAISY (4));
```

You show that two or more FILES are declared together by enclosing the

names of the FILES in parentheses, separated by commas. The GROUPS DONALD and DAISY and the other child variables below them then need to be declared only once, and they will still be made part of both FILES, with the same names, repetition counts, and other arguments.

The advantage of using this shortcut is that you have to describe a particular complex data FILE structure only once, although you can use it again as part of a different FILE. (Because two FILES have the same data structure does not mean that they need to contain the same data.) The difficulty is that you create several variables with the same name, which causes ambiguity. Back in Section 4.6, we described how variables with the same names from different FILES must be distinguished later in assertion statements by giving them a prefix of the name of their parent file, for example, MICKEY.DONALD and PLUTO.DONALD.) There's no real problem as long as you don't try to put more than one variable with the same name in the same FILE.

You may also find it useful to delineate the same data structure as part of two or more different FILES, without having the whole FILES be the same. You can do this by naming the same variable or variables as descendent(s) of variables from the different FILES. This is easy to do in the first form of MODEL data declaration syntax (impossible in the second), because the immediate descendents of a variable are listed at the same time a variable is declared. As an example, suppose G3 and H2 are GROUPS in different FILES F1 and F2, and you wish them both to include the GROUPS ABLE and BAKER. To do this you could write

```
F1 IS FILE (G1, G2, G3);
          .
          .
          .
G3 IS GROUP (ABLE, BAKER);
ABLE IS GROUP (CAT, DOG);
BAKER IS GROUP (BIRD, FISH);
          .
          .
          .
```

to describe the first FILE F1. Elsewhere in the specification you could write

```
F2 IS FILE (H1, H2, H3);
          .
          .
          .
H2 IS GROUP (ABLE, BAKER);
          .
          .
          .
```

to describe the second FILE F2. When you do this, the tree structures ABLE and BAKER described in FILE F1, will be included in FILE F2 without having to declare them a second time. The same thing could be done if FILE F1 were written in the second form of MODEL syntax, as follows:

```
1  F1 IS FILE,
          .
          .
          .
   2  G3 IS GROUP,
      3  ABLE IS GROUP,
      3  BAKER IS GROUP,
          .
          .
          .
```

In either case, you will need to use FILE name prefixes in your assertions to disambiguate the shared variable names.

You can also create parallel FIELD variables of the same data type and length in the same declaration, as in:

```
(CASPER, WENDY) ARE FIELDS (PIC 'ZZ9V.99');
```

Their names will be sufficient to distinguish them from each other, within a single FILE. (You could achieve the same effect just as easily using subscripts, that is, declaring CASPER(I), with I having a range of two.) However, you can't declare multiple RECORDS or GROUPS at one time. That's because the data structures under these RECORDS or GROUPS will be made the same, including the names of FIELD variables. You'll end up having two FIELDS with the same name in the same FILE, one under each RECORD, so that you can't use the name of the parent FILE as a prefix to disambiguate them. So, if you write an assertion statement defining a value for one of these FIELDS, it will be unclear which one you meant.

5.3.3 DECLARING REPETITION COUNTS AND OPTIONAL DATA STRUCTURES

In MODEL data declaration you have four options for expressing the repetition counts of GROUPS, RECORDS, and FIELDS. These options apply equally well to data declaration using either of the syntaxes we discussed previously. (Remember that in the first syntax, repetition count information is given as part of the declaration of the immediate parent of a particular variable, while in the second syntax it is given as part of the declaration of the variable itself.) These options are as follows:

1) If a variable has only one repetition, then you can omit the repetition count altogether.

2) If you know how many repetitions a particular variable will have, then you can enter that number as the repetition count directly. So, to indicate how many STUDENT FIELDS there were for each TEST_SCORE RECORD in the data declaration of the specification EXAMPLE given in Figure 5.2, we wrote

```
TEST_SCORE IS RECORD (STUDENT (12));
```

using the first syntax or

```
3  STUDENT(12) IS FIELD (PIC '99');
```

using the second.

3) If you're totally unsure how many repetitions a particular variable will have, then you can use an asterisk in place of a number for the repetition count, as in

```
DEPT IS FILE (PRODUCT (*), EMPLOYEE (*));
```

Whenever you use an asterisk in place of a repetition count, then the

MODEL system will try to optimize the use of memory space. If optimization is not possible and all the repetitions of a variable have to be in memory, then space is directly available for up to 9999 repetitions. You will then get a warning message in your error report. Giving more specific information, if you know it, will save space, and will allow higher repetition counts.

4) You can also enter a minimum-maximum range on the number of expected repetitions of a variable, if you have some knowledge of what the data will look like. This method may be your best compromise in saving space. An example of using a repetition range, (modifying the above) would be

```
DEPT IS FILE (PRODUCT(1:8), EMPLOYEE(4:9));
```

An example of a single statement illustrating all the above options for expressing repetition count is

```
ADDAMS IS GROUP (GOMEZ, MORTICIA(5), WEDNESDAY(3:6), PUGSLEY(*));
```

If you decide to use either an asterisk or a minimum-maximum range when declaring the repetition count of a variable, you must be sure that your specification and data provide an alternative information source to the MODEL compiler for specifying that variable's range. Examples of such sources of information might be ENDFILE markers, range propagation, or control variables with a SIZE or END prefix. See Sections 7.3, 8.2 and 8.3 for detailed discussion of these options.

When you use an asterisk or give a repetition range whose minimum is 0, you are defining a particular optional data structure. This means that the relevant variable and all its child variables (if it is not a FIELD) may or may not exist as part of the data tree, depending on the SOURCE data.

```
MARX_BROTHERS IS GROUP (GROUCHO, CHICO, HARPO, ZEPPO (0:5));
```

For example, in the above SOURCE FILE declaration statement declaring the FIELD descendents of a GROUP, if only three FIELDS are included in the SOURCE data, then ZEPPO will left out of the MARX_BROTHERS.

5.3.4 DECLARING INTERIM DATA STRUCTURES

You may find it useful to declare interim data structures in two situations. The first is if it is difficult for you to directly define TARGET FILE variables in terms of SOURCE FILE variables in a single expression. For example, in the sample specification introduced in Chapter 3 (Figure 3.1), in order to calculate standard deviations, we had to add up the squared differences of each student's test score from the mean of the scores on that test. Then we had to take the mean of these squared differences and find its square root (the relevant assertions are shown below).

```
S_TEST(I,J) = IF J > 1
   THEN S_TEST(I,J-1) + ((STUDENT(I,J) - MEAN_TEST(I))**2)
   ELSE ((STUDENT(I,J) - MEAN+TEST(I))**2);

STD_TEST(I) = (S_TEST(I,12)/12)**.5;
```

This would have been difficult to do in one assertion, but it was easy to do in two using the interim data structure below.

```
1 INT IS FILE,
   2 INT_TEST(3) IS GROUP,
      3 (M_TEST,S_TEST)(12) IS FIELD (PIC '9999V.9999');
```

We defined the accumulating sums of the squared differences as equal to the interim FIELD S_TEST(I,J). We were then able to use the interim total to directly define the value of the TARGET FILE variable.

The second situation when you would want to declare an interim data structure is as a convenience. You can define an interim variable as equaling a complex or cumbersome expression that would appear in several assertion statements. Instead of typing the expression over and over, you can then use the interim variable name to stand for it.

In MODEL interim GROUP, RECORD, or FIELD variables, do not need to be declared as part of FILES and may be declared separately. Also interim GROUPS and FIELDS do not need to be declared as part of RECORDS. (The MODEL system will fill in the higher levels of these interim data structures automatically.) However, the top level of an interim data structure, like a FILE, must have a repetition count of one. Also if your interim data structure is declared using the second form of data declaration syntax, then the top level must be given a level number of 1 as in the above example.

5.4 FILE DECLARATION STATEMENTS

All external data must be declared as part of SOURCE or TARGET FILES. SOURCE, TARGET, and interim FILES are all declared the same way. Figure 5.5 shows the syntax for FILE declaration using the first type of MODEL syntax (demonstrated in Figure 5.2 (a)). In this form of syntax, the names and repetition counts of the immediate descendents of the FILE are listed in the FILE declaration statement, as in

```
POPEYE IS FILE (BLUTO(2), OLIVE(3));
```

A FILE declaration statement also gives the name of the FILE and optionally describes the FILE'S organization. A FILE'S organization refers to whether the FILE uses a sequential access method (SAM) or an index sequential access method (ISAM) using KEYS. KEYS in ISAM FILES act like catologue numbers in a library which allow particular books to be found without having to search through all the shelves (see below). Alternatively with SAM FILES, the computer has to search from the beginning of the FILE, RECORD by RECORD, until the appropriate RECORD is

found.

```
1  <FILE declaration statement> ::= <variable> [<IS>] <FILE>
                                    <FILE argument>;
   2  <FILE> ::= FILE ¦ FILES
   2  <FILE argument> ::= <descendents> [<FILE description>]
                          [<storage description>]
      3  <FILE description> ::= [STORAGE [NAME] [<IS>] <name>] [KEY [<FILE
                                 name>] [<IS>] <FIELD name>]
      3  <storage description> ::= [DEVICE [<IS>] DISK] [ORG[ANIZATION]
                                    [<IS>] SAM ¦ SEQUENTIAL ¦ ISAM ¦
                                    INDEX_SEQUENTIAL]
```

Figure 5.5

Syntax of the FILE Declaration Statement

Figure 5.6 presents a specification using the keyed ISAM FILE GINGER as both SOURCE and TARGET. This specification is described in detail over the next several paragraphs to explain how KEYS and ISAM FILES are used. The specification contains two FILE declarations and two assertions. The ISAM FILE GINGER is declared using the following statement:

```
GINGER IS FILE (MRS_HOWELL) KEY IS MINNOW ORG IS ISAM;
```

This statement says that the FILE GINGER is declared to have an index sequential organization, and that it contains a set of RECORDS named MRS_HOWELL, each containing an identifying KEY FIELD called MINNOW. Only one RECORD variable may be declared in an ISAM FILE like GINGER, and it must have a repetition count of one. (The reason why is explained below.) The RECORD variable MRS_HOWELL in the ISAM FILE GINGER is declared to contain two FIELDS: the KEY FIELD MINNOW and a second FIELD called PROFESSOR which holds data.

```
MODULE: GILLIGAN;
SOURCE: SKIPPER,GINGER;
TARGET: GINGER;

SKIPPER IS FILE (MARY_ANN);
   MARY_ANN IS RECORD (MR_HOWELL(*));
      MR_HOWELL IS FIELD (PIC '9');

GINGER IS FILE (MRS_HOWELL) KEY IS MINNOW ORG IS ISAM;
   MRS_HOWELL IS RECORD (PROFESSOR, MINNOW);
      PROFESSOR IS FIELD (PIC 'Z99');
      MINNOW IS FIELD (PIC '9');

I IS SUBSCRIPT;

POINTER.MRS_HOWELL(I) = MR_HOWELL(I);
NEW.PROFESSOR(I) = OLD.PROFESSOR(I) + 2;
```

Figure 5.2

Using the ISAM FILE GINGER as both SOURCE and TARGET

KEYS are optional (and infrequent) in the declaration of SAM FILES, but mandatory in the declaration of ISAM FILES like GINGER. The content of a KEY FIELD like MINNOW is a number or alphanumeric which uniquely identifies each keyed RECORD in the FILE. (Keyed RECORDS in SAM FILES must be presorted in ascending order according to their KEY values.) Each RECORD in a keyed FILE must be declared to have one KEY with fixed position. This means that the number and lengths of the FIELDS to the left of the KEY FIELD must be exactly specified at the time of data declaration. In other words, the FIELD PROFESSOR, located to the left of the KEY FIELD MINNOW, cannot have an unspecified length. This insures that the same FIELD is located as the KEY for each RECORD. (The number and lengths of FIELDS to the right of each KEY FIELD, if any, need not be specified in data declaration.) Additionally, you cannot have more than one keyed RECORD variable in the same FILE, and you cannot refer to keyed RECORDS by name in assertion statements. Also, you cannot use the NEXT.X control variable (Section 8.6) to sort keyed RECORDS into GROUPS, as you could for unkeyed FILES.

As stated above, keyed ISAM FILES are declared as having only one RECORD variable with no repetition count. This does not mean that an ISAM FILE must contain only one RECORD. Instead, the data structure for the keyed RECORDS of the ISAM FILE is specified in an assertion

statement defining a POINTER control variable (see Section 8.8 and below). Such an assertion must be part of every specification containing a keyed FILE, whether SAM or ISAM.

An assertion defining a POINTER control variable is used to select and order certain of the RECORDS from the keyed FILE, based on a match of KEY FIELDS in the RECORDS with POINTER FIELDS from a separate reference FILE (also called a transaction FILE). In the case of our specification, the reference FILE, named SKIPPER, contains a vector of POINTER FIELDS named MR_HOWELL. The POINTER FIELD vector MR_HOWELL(I) contains the same identifying numbers used in the KEY FIELD MINNOW from the ISAM FILE GINGER, but not necessarily every one, and not necessarily in the same order.

When we write the assertion statement

```
POINTER.MRS_HOWELL(I) = MR_HOWELL(I);
```

we show that we want the RECORDS (MRS_HOWELL) of the keyed FILE to be given the same data structure as the POINTER FIELDS MR_HOWELL from the FILE SKIPPER. The RECORDS of the ISAM FILE will be ordered, by matching KEY to POINTER, so that they are in the same order as the POINTERS. The RECORDS (MRS_HOWELL(I)) of the keyed FILE GINGER will also be organized as a vector, because this is the organization of the POINTER FIELD MR_HOWELL(I).

We then use the assertion

```
NEW.PROFESSOR(I) = OLD.PROFESSOR(I) + 2;
```

to show that we wish to redefine the value of the FIELD PROFESSOR contained in the RECORDS whose KEYS were listed in MR_HOWELL(I). OLD.PROFESSOR indicates the original value of the FIELD PROFESSOR obtained from the original (SOURCE) FILE GINGER. NEW.PROFESSOR indicates the value of the FIELD after updating. (The use of the

prefixes OLD and NEW is explained in Section 4.6.) Only those RECORDS containing KEYS corresponding to the POINTERS from the reference FILE SKIPPER will be updated before being placed in ISAM TARGET FILE GINGER. RECORDS containing a KEY FIELD for which there is no corresponding POINTER will not be updated before being entered into the TARGET FILE. (Without an assertion to the contrary, the MODEL compiler assumes that the value of the KEY FIELD MINNOW in SOURCE FILE GINGER and TARGET FILE GINGER will be the same.)

The result is that, by defining the appropriate set of POINTERS, you can create a TARGET FILE containing a subset of updated RECORDS from an ISAM SOURCE FILE, reordered any way you choose, in any number of dimensions, so that referencing particular RECORDS will be made easy. The flexibility and speed of accessing RECORDS in keyed FILES is counterbalanced by the greater effort required on your part to set them up. Whether this effort is justified depends on the particular problem you wish to solve.

The ‹storage description› syntax element in the FILE declaration syntax diagram can be optionally expanded to include more information about external devices such as magnetic tape. The syntax for the expansion is given in Figure 5.7, although the majority of users will never need to use it. (Disk storage is the default choice if no information about external storage is included in your FILE declaration statement.) If the storage description syntax is not self-explanatory, additional information may be found in the PL/1 Programmer Reference Manual.

```
1  <storage description> :: = DEVICE [<IS>] CARD ¦
                              PRINTER [[( ] LINE = <number of positions> [>]] ¦
                              TAPE [( ]<tape description>[ )] ¦
                              DISK [( ]<disk description>[ )] ¦
                              TERMINAL [( ]UNIT [=] <unit number>[ )] ¦
                              PUNCH ¦ INT[ERNAL]
                              [[,] MODEL [=] <manufacturer and model>
                              [[,] SYSTEM [=] <operating system>
   2  <tape description> ::= [[,] VOL[UME] [=] <internal name>]
                              [[,] TRACKS [=] 7 ¦ 9]
                              [[,] PARITY [=] ODD ¦ EVEN]
                              [[,] DENSITY [=] 200 ¦ 556 ¦ 800 ¦ 1600 ¦ 6250]
                              [[,] LABEL [=] IBM_STD ¦ ANSI_STD ¦ NONE ¦
                              BYPASS]
                              [[,] START [=] <FILE sequence on tape>]
                              [[,] CODE [=] EBCDIC ¦ BCD ¦ ASCII]
   2  <disk description> :: = [ORG[ANIZATION] [<IS>] SAM ¦ SEQUENTIAL ¦ ISAM ¦
                              INDEX_SEQUENTIAL]
                              [,] <RECORD format>
                              [[,] UNIT [=] 2305 ¦ 2311 ¦ 2314 ¦ 330 ¦
                              3330_1]
                              [[,] CYLINDERS [=] <integer>[,<integer>]*]
      3  <RECORD format> ::= FORMAT [<IS>] FIXED ¦ VARIABLE ¦
                              VARIABLE_SPANNED ¦ UNDEFINED
                              [,] [MAX_BLOCKSIZE [=] <number pf bytes>
                              [,] [MAX_RECORDSIZE [=] <number of bytes>
```

Figure 5.7

Syntax of Storage Description

5.5 GROUP AND RECORD DECLARATION STATEMENTS

GROUPS and RECORDS are the intermediate level of the data tree, having a substructures below them. RECORDS are special because they are the unit of physical transfer of information between internal and external data storage. GROUPS are intermediate data structures which are not RECORDS. GROUPS can be above or below RECORDS in the data tree, for example

```
ROLF IS GROUP (HANS(3));        ¦       CLAUDIA IS RECORD (ANNA(4));
HANS IS RECORD (OTTO(3));       ¦       ANNA IS GROUP (URSULA(2));
```

GROUPS can be above or below other GROUPS, for example

```
TONY IS GROUP (MARIA(2));
MARIA IS GROUP (BERNARDO(7));
```

Each FIELD or piece of data, except for interim FIELDS, must have only one RECORD above it in the FILE. (Interim FIELDS need not be in RECORDS because they are not transferred to or from secondary storage.) However the same FIELD can simultaneously be a member of many GROUPS, one above the other, for example

```
TONY IS GROUP (MARIA(2));
MARIA IS GROUP (BERNARDO(7));
BERNARDO IS FIELD (PIC 'ZZ9');
```

The syntax for declaring GROUPS and RECORDS is very similar to that for declaring FILES. The major difference is that the ‹FILE description› and ‹storage description› syntax elements are omitted. GROUPS and RECORDS, as demonstrated above, are declared the same way. The syntax diagram of GROUP or RECORD declaration statements is given in Figure 5.8

```
1  ‹GROUP or RECORD declaration statement› ::= ‹Variable› [‹IS›] ‹GROUP› ¦
                                               ‹RECORD› ‹descendents›;
   2  ‹GROUP› ::= GROUP¦GROUPS¦GRP¦GRPS
   2  ‹RECORD› ::= RECORD¦RECORDS¦REC¦RECS
```

Figure 5.8

Syntax of the GROUP or RECORD Declaration Statement

5.6 FIELD DECLARATION STATEMENTS AND DATA TYPES

FIELDS are the parts of the data tree which hold the values of individual pieces of data. (The values of the FIELDS making up TARGET FILES are defined in assertions, as explained in the next two chapters.) Each FIELD contains a particular kind of data, as determined by the FIELD's data type. Here we will explain the characteristics of each data type and examine how to use MODEL syntax to express them.

FIELDS are different from GROUPS and RECORDS because they take no descendents, since they are at the lowest level of the data tree. The syntax diagram of the FIELD declaration statement also contains a ‹data type definition› syntax element which is unique to FIELDS (see Figure 5.9). This element gives the FIELD's data type. There are six main data types which can be used in MODEL FIELDS: character string, bit string, numeric string, decimal, binary,and picture. The decimal and binary data types further divide into fixed and floating subtypes. Each of these data types will be explained in turn.

```
1  ‹FIELD declaration Statement› ::= ‹variable› [‹IS›] ‹FIELD›
                                     [(] ‹data type definition› [)];
   2  ‹FIELD› ::= FIELD¦FIELDS¦FLD¦FLDS
   2  ‹data type definition› ::= ‹character string› ¦ ‹bit string› ¦
                                 ‹decimal› ¦ ‹binary› ¦ ‹picture›
      3  ‹character string› ::= CHAR[ACTER] ‹string format›
         4  ‹string format› ::=  [(] ‹no. of elements› ¦ ‹minimum no. of
                                 elements› : ‹maximum no. of elements› [)]
            5  ‹no. of elements› ::= ‹unsigned integer›
            5  ‹minimum no. of elements› ::= ‹unsigned integer›
            5  ‹maximum no. of elements› ::= ‹unsigned integer›
      3  ‹‹bit string› ::= BIT [(] ‹no. of elements› [)]
      3  ‹numeric string› ::= NUM [(] ‹no. of elements› [)]
      3  ‹decimal› ::= DEC[IMAL]  ‹fixed format› ¦ ‹floating format›
         4  ‹fixed format› ::= [FIX[ED]] [(] ‹no. of elements› [)]
         4  ‹floating format› ::= FLOAT [(] ‹precision› [)]
            5  ‹precision› ::= ‹unsigned integer›
      3  ‹binary› ::= BIN[ARY]  ‹fixed format› ¦ ‹floating format›
      3  ‹picture› ::= PIC[TURE] '[[(‹no. of repetitions›)]
                        9¦Z¦*¦Y¦.¦,¦/¦B¦S¦+¦-¦$¦T¦I¦R¦E¦X¦A ]*'
         4  ‹no. of repetitions› ::= ‹unsigned integer›
```

Figure 5.9

Syntax of the FIELD Declaration Statement

Following PL/1, data types used in MODEL fall into two classes, printable and nonprintable. Printable data types, that is, character string, numeric string, and picture, can be read as TARGET data in the form of conventional characters and digits. They can also be entered this way as raw SOURCE data. (That's why we only used the picture data type in the sample specification EXAMPLE in Chapter 3.)

On the other hand, nonprintable data types, such as decimal, binary, and bit, are represented by the computer in a more compact form which takes up less room in memory. If you try to output information in this form, you will see what appears to be a mixture of random characters, which you won't be able to read. Also when entering raw data, you shouldn't use nonprintable data types in your SOURCE FILE declaration, because the system won't be able to read it. However, you will still want to use these data types for certain applications, such as doing many routine calculations, because they can be processed much more quickly than the printable data types. You can use nonprintable data types to produce compact data FILES for storage on tape or disk, or for direct input into another program or specification which is written to accept SOURCE data in this form.

Manually entered data and printed reports should use picture, numeric, or character string data types. You can use nonprintable data types in interim FIELDS by declaring them to be decimal, bit, or binary. You can then write assertion statement(s), defining the nonprintable interim FIELDS in terms of printable SOURCE FILE variables, so that the conversion is done automatically. To get readable output, you can then use these interim variables to define TARGET FILE variables declared with printable data types.

5.6.1 CHARACTER STRING VARIABLES

Character string variables are just that, strings of characters. They can be made up of combinations of any characters which your keyboard will print, including combinations of numbers and letters. Character string variables, like character string constants, are used with the concatenation operator (Section 4.8) and string functions (Section 4.10 and the appendix) to create string expressions (Section 4.9). (However, unlike character string constants, you should not use apostrophes to surround character string SOURCE data, unless you want the apostrophes to be read as part of a variable.)

When you declare a character string variable, the MODEL system needs you to specify its length. You have two options for doing this. One option is to declare a specific number of characters that you expect your character string variable to be, as in,

```
WANDA IS FIELD (CHARACTER (10));
```

(If your character string variable is repeating, then all its members must be the length you declare.) The other alternative is to enter a minimum and maximum expected length for the character string variable, as in,

```
GLENDA IS FIELD (CHAR (3:7));
```

You must then include an assertion statement to define the length of the character string variable in terms of a control variable with an LEN prefix (see Section 8.4). For example to specifically define the length of GLENDA you could write

```
LEN.GLENDA = 5
```

5.6.2 BIT STRING VARIABLES

Bit string variables are very similar to bit string constants (Section 4.7.2). They are used with logical operators (Section 4.8) as operands in logical expressions (Section 4.9.4). They can also be used in string expressions like character string variables. Unlike bit string constants, bit string variables must be base 2. Unlike character string variables,they cannot be of unspecified length. An example of a bit string declaration is

```
HENRY IS FIELD (BIT (3));
```

5.6.3 NUMERIC STRING VARIABLES

Although numeric string variables are called strings, they are a form of arithmetic variable, used in arithmetic expressions (Section 4.9.3) with arithmetic operators (Section 4.8). (Numeric string is the only MODEL data type which is not also present in PL/1.) Numeric string variables are unsigned integers. When they are declared, you should specify the number of expected digits, as in

```
5  HERO (3) IS FIELD (NUM (6));
```

5.6.4 DECIMAL AND BINARY VARIABLES

Decimal and binary variables are arithmetic variables, also used in arithmetic expressions (Section 4.9.3) with arithmetic operators (Section 4.8). These types also allow you to use a "floating point" feature (which we shall describe shortly), which allows your specification to handle very large or very small numbers, which would ordinarily have many digits to the right or left of the decimal point. The binary data type is preferred to the decimal when you want to perform complex arithmetic computations. That's because binary data is stored more compactly in the memory of the computer than decimal data. Also arithmetic operations are more efficient using binary operands.

Both decimal and binary data types can be expressed in a fixed or floating format. In fixed format, you tell the computer how many digits you expect there to be in your data variable. For example,

```
6  LISA (4) IS FIELD (BINARY FIX (5));
```

tells the computer that LISA consists of four five-digit base 2 numbers.

Decimal and binary data can also be expressed in floating format. In this case the computer keeps track of where the decimal point should be placed for each piece of data. In floating format, data is expressed

as a number, containing a certain number of digits, multiplied by 10, for decimal, or 2, for binary, raised to a particular power. This data type can save a lot of space in the computer's memory if the number to be represented is very large or very small.

In floating format notation, the number itself is called the mantissa. The number of digits in the mantissa is called its precision. The precision represents the number of digits (called significant digits) that you want the computer to keep track of, and you specify it as part of the declaration of each floating point variable. You can specify a maximum precision of 34 for floating point decimal and 113 for floating point binary. An example of a legal floating point declaration would be

```
STAN IS (DECIMAL FLOAT (25));
```

When doing calculations if the result has more significant digits than the precision you specified, then the computer will round the result off.

5.6.5 PICTURE VARIABLES

The picture data type is the most common one to use for processing arithmetic data. Picture variables are used in arithmetic expressions (Section 4.9.3) with arithmetic operators (Section 4.8). You declare a picture variable in terms of a series of symbols through which you can specify what characters will be allowed in each position in the variable. A Z means that a leading zero should be omitted and printed as a blank, an S holds a space for a positive or negative sign, an A represents a space where a letter can be inserted, and so on (see Figure 5.10 below).

By putting these symbols in the proper order, you can exercise considerable control over how your data will be read and printed. For example, in the sample specification in Chapter 3, the means and standard deviations in the TARGET FILE are declared as follows:

```
(MEAN_TEST,STD_TEST) IS FIELD (PIC 'ZZZ9V.9999');
```

The 9's here stand for decimal digits. The Z's also stand for digits, but mean that leading 0's should be printed as blanks. The period shows where the decimal point is positioned, and the V shows that it should be printed as part of the number. IN total, ZZZ9V.9999 means that there are four possible digits to the left and to the right of the decimal point, and that leading 0's won't be printed.

The symbols used for picture in the author's implementation of MODEL are explained in Figure 5.10. Your system may allow fewer or different symbols. Therefore, you may want to check the User's Guide for the version of PL/1 implemented on your system.

X stands for any character. A picture variable of all X's is just like a character string variable.
A stands for any alphabetic character or a blank character.
9 stands for a decimal digit in a given position.
Z also stands for a decimal digit, except that zeros on the left will not be printed. For example, 0064 would be printed as 64 if the field were declared as ZZZZ.
* also stands for a decimal digit, but the leading zero is replaced with an asterisk instead of being omitted. Hence **64.
Y stands for a decimal digit, except that a spac will be printed for any zero in any position.
(n) can be used to indicate a number of repetitions of the following character. For example, (3)9 indicates 3 decimal digits. In some versions of PL/1 it may cause problems if you try to show that a character is repeating 10 or more times.
T stands for a digit or a plus sign or minus sign, if there is one.
I is the same, except that only a plus sign will be printed. Negative numbers will be printed without a minus sign.
R is the same, except that it prints a minus when the number is negative, but no plus when it is positive or zero.
. indicates the position in which a variable's decimal point is expected to appear.
V indicates the position where you would like a decimal point to be placed when a string of numbers is read, without your having to actually type the decimal point in. If no V character is used, this tells the computer that the decimal point is on the right; that is, the picture variable is seen as an integer. If you use V followed by a period, ".", this will cause a decimal point to be inserted, and later printed out as part of the variable.
, is the position for a comma to be inserted.
$ is the position for the dollar sign. When leading zeros are omitted, the dollar sign will be moved up next to the leftmost printed digit. This symbol cannot occur in the middle of a field.
+ is the position for a plus sign. This is like I except that the sign is placed next to the leftmost digit. This symbol cannot be used in the middle of a field.
- is the same, for a minus sign, in case the number is negative. This symbol cannot occur in the middle of a field.
S will print either sign, like T, but next to the leftmost digit. This symbol cannot occur in the middle of a field.
E indicates the position of the exponent in a floating point number. For example, 4E3 is 4 time ten to the third, or 4,000. Likewise, 5E-4 is 5 time 10 to the minus fourth, or .0005.

Figure 5.10

Symbols Used in the Picture Data Type

CHAPTER 6

TYPES OF ASSERTION STATEMENTS

6.1 OVERVIEW

Assertion statements are equations. They define all the FIELD variables making up your interim and TARGET data structures. (The use of assertions with interim data structures is explained in Section 5.3.4.) Each assertion statement defines one dependent variable placed on the left-hand-side of an equals sign. This variable is defined as equal to an expression on the right-hand-side composed of constants, operators, variables, and functions.

As stated previously, in MODEL each variable can have only one value. Therefore each FIELD to be defined with a different value has to have a unique name or distinguishing subscript. Since MODEL is nonprocedural, you don't have to worry about the order in which your variables are defined. You just have to make sure to write an assertion statement to define every field in your TARGET and interim data structures.

In some cases you can optionally omit subscripts when writing assertions. (The conditions for subscript omission, as well as other aspects of using subscripts in assertions, will be explained in the next chapter.) This is meant to be a shortcut in writing rather than in thinking. Keep in mind that subscripts are implied, when you write assertions using variables with omitted subscripts.

The objective of this Chapter is to present the syntaxes of the different types of assertion statements along with examples. Assertion statements in MODEL may be either simple or conditional, as shown by the

following syntax diagram:

‹assertion statement› ::= ‹simple assertion statement› ¦ ‹conditional assertion statement›

A simple assertion statement defines the value of the dependent variable on the left side of the equation in terms of a single expression on the right side. An example is

```
A = B + 3;
```

Conditional assertion statements define a dependent variable as equal to one of several expressions, depending on the value of a condition. An example of a conditional assertion statement is

```
A = IF B > 2 THEN 7 ELSE 12;
```

In the previous Chapter we explained how you have the option of leaving certain attributes of your SOURCE data, such as variable ranges, unspecified in data declaration. You must then define these attributes in your assertion statements using control variables (introduced in Section 4.6). For example, suppose you declared a FIELD variable ALBATROSS to have an uncertain number of repetitions, as follows

```
PENGUIN IS GROUP (ALBATROSS(*), SEAGULL);
```

You could then define the range of ALBATROSS in an assertion statement, such as

```
SIZE.ALBATROSS = 2;
```

Control variables, such as SIZE.ALBATROSS, may then be used as independent variables in defining other variables in other assertions. The uses of the various types of control variables will be explained in the Chapter 8.

6.2 SIMPLE ASSERTION STATEMENTS

As stated previously, assertion statements in MODEL are of two kinds, simple and conditional. Simple assertion statements are called simple because they contain only one expression to define the dependent variable. This expression may be a logical, arithmetic, Boolean, string, or comparison expression, as described in Chapter 4. The syntax diagram for simple assertions is shown in Figure 6.1.

```
1  <simple assertion statement> ::= <subscripted variable> = <any expression>;
   2  <subscripted variable> ::=   <name> [(<subscript expression>
                                   [, <subscript expression>]*)] |
                                   (<name> [(<subscript expression>
                                   [, <subscript expression>]*)]
                                   [,<name> [(<subscript expression>
                                   [, <subscript expression>]*)]])
      3  <subscript expression> ::= <arithmetic expression>
   2  <any expression> ::= <logical expression> | <arithmetic expression> |
                           <string expression> | <Boolean expression> |
                           <comparison expression>
```

Figure 6.1

Syntax of Simple Assertion Statements

The dependent variable in a simple assertion statement may take subscripts, and may be a qualified name variable. (The use of subcript expressions is described in detail in the next chapter.) Two or more dependent variables can also be defined in one assertion statement, as long as each receives the same definition. In that case, the list of variable names must be enclosed in parentheses, with individual names separated by commas. You can define the the dependent variable in terms of an expression containing qualified name variables, constants, non-qualified variables (with or without subscripts), and functions.

Some examples of legal simple assertion statements are as follows:

```
A = B + 5;
SIZE.JLA = 12;
X(I,J) = 4 * I**J;
(JUNG,FREUD) = ADLER;
```

6.3 CONDITIONAL ASSERTION STATEMENTS

6.3.1 OVERVIEW

Conditional assertion statements are more complicated than simple assertion statements, because the choice of defining expression depends on the value of a condition. The dependent variable in a conditional assertion statement is defined in terms of one expression if the condition is true, and in terms of another if the condition is false. This condition is a Boolean expression. A Boolean expression, as defined in Section 4.9, is an expression which has a binary truth value of true or false. For example, the comparison (Boolean) expression "A > 7" must be either true or false, when A is an arithmetic variable.

The process works as follows. In the conditional assertion statement

```
MONTANA = IF A > 7 THEN 5 ELSE 12;
```

the dependent variable MONTANA will be defined as equalling 5 if the condition A > 7 is true, or as 12 if the condition is false. The part of the above statement containing the keyword IF, followed by the conditional Boolean expression, is called the IF-clause. The first defining expression for the dependent variable in a conditional assertion statement, 5 in the above case, is preceded by the keyword THEN. The second defining expression, 12 above, is preceded by the keyword ELSE. A more complex conditional assertion statement may contain more than one IF-clause and more than two defining expressions (see below).

The MODEL language allows two alternative syntaxes to express conditional assertions. They are based on the syntaxes of statements in the PL/1 and ALGOL computer languages. They differ in whether the condition precedes the dependent variable on the left-hand-side of the assertion, or the defining expression on the right-hand-side. The effect of writing conditional assertions using either syntax is the same.

The ALGOL-like form, with the condition on the right-hand-side, is more terse and more closely resembles an algebraic equation. An example would be the assertion

```
SARAH = IF JULIA = 17 THEN 3 ELSE 2;
```

The syntax diagram for this form is shown in Figure 6.2.

```
1  <conditional assertion statement> ::= <subscripted variable> =
                                         <conditional expression>;
   2  <conditional expression> ::= <IF-clause> THEN <conditional>
                                   [ ELSE <conditional> ]
      3  <IF-clause> ::= IF <Boolean expression>
      3  <conditional> ::= <conditional expression> ¦ <any expression>
```

Figure 6.2

Syntax of ALGOL-like form of Conditional Assertion

The PL/1-like syntax for conditional assertion statements starts with the condition. Therefore, the example from above would be rewritten as

```
IF JULIA = 17 THEN SARAH = 3; ELSE SARAH = 2;
```

In this syntax you repeat the name of the dependent variable, accompanied by an equals sign, in front of each defining expression, and follow it with a semi-colon, as if it were a separate assertion. However, unlike separate assertions, each nested assertion in a Pl/1-form conditional assertion statement must have the same dependent variable. The syntax diagram for this form is shown in Figure 6.3.

```
1  <conditional assertion statement> ::= <IF-clause> THEN
                                         <simple assertion statement>;
                                         [ELSE <assertion statement>];
```

Figure 6.3

Syntax of PL/1-like form of Conditional Assertion

To provide further clarification, Figure 6.4 shows the conditional assertions from the sample specification EXAMPLE written in both forms of syntax for comparison purposes. These assertions demonstrate one of the most common uses of conditions in assertion statements, that is, to differentially define the members of a series depending on their subscripts. The fact that this can be accomplished in a single statement demonstrates the power of the MODEL language to express complex ideas in simple manner.

```
M_TEST(I,J) = IF J > 1
                THEN M_TEST(I,J-1) + STUDENT(I,J)
                ELSE STUDENT(I,J);

S_TEST(I,J) = IF J > 1
                THEN S_TEST(I,J-1) + ((STUDENT(I,J) - MEAN_TEST(I))**2)
                ELSE ((STUDENT(I,J) - MEAN_TEST(I))**2);
```

ALGOL-like Syntax for Conditional Assertions

```
IF J > 1
THEN M_TEST(I,J) = M_TEST(I,J-1) + STUDENT(I,J);
ELSE M_TEST(I,J) = STUDENT(I,J);

IF J > 1
THEN S_TEST(I,J) = S_TEST(I,J-1) + ((STUDENT(I,J) - MEAN_TEST(I))**2);
ELSE S_TEST(I,J) = ((STUDENT(I,J) - MEAN_TEST(I))**2);
```

PL/1-like Syntax for Conditional Assertions

Figure 6.4

Alternate Syntaxes for Conditional Assertions

Both forms of syntax for conditional assertion statements are written so that the second defining expression, starting with ELSE, is optional and may be omitted. This means it is perfectly legal to write an assertion like

```
URANUS = IF JUPITER(I) > SATURN(I) THEN 15;
```

If the condition is true, then the dependent variable will be defined in terms of the expression starting with THEN, as described previously. If

the condition is false, then the assertion will not be used, and no definition will be given to the dependent variable. When a conditional assertion statement like the above (without an ELSE) is used to define a scalar, you must be sure to choose a condition that will be true only once. If you don't, then two or more values will be used to define the same scalar, causing your program to bomb.

6.3.2 NESTED CONDITIONAL ASSERTIONS

As stated previously, both forms of syntax for conditional assertions allow you to write assertions containing more than one condition and more than two defining expressions. Suppose, for example, we wanted to write a specification to keep track of how much we should pay various employees in our company. We can use the variable PAY(I) to keep track of paychecks, with the subscript I referring to each individual by their number on the payroll. Amount of pay to each person depends on two factors, job-type and number of hours worked, which we can call JOB(I) and HOURS(I), respectively. We will also assume that there are three possible types of jobs in our company: executive vice-president in charge of advertising, programmer-analyst, and janitor. We can write an assertion to define a pay amount for each individual (in terms of what they're worth) as follows:

```
PAY(I) = IF JOB(I) = 'PROGRAMMER-ANALYST'
         THEN 50.00*HOURS(I)
         ELSE IF JOB(I) = 'JANITOR'
              THEN 12.50*HOURS(I)
              ELSE 2.17*HOURS(I);
```

That the executive vice-president in charge of advertising should be the one making $2.17/hour is understood. (It's the only job-type left.)

In terms of the PL/1 form of syntax for conditional assertion statements, the above assertion would be rewritten as

```
IF JOB( I ) = 'PROGRAMMER-ANALYST'
THEN PAY( I ) = 50.00*HOURS( I );
ELSE IF JOB( I ) = 'JANITOR'
     THEN PAY( I ) = 12.50*HOURS( I );
     ELSE PAY( I ) = 2.17*HOURS( I );
```

The above example, written in both syntaxes shows the nesting of an additional condition and two additional defining expressions within the ELSE part of a conditional assertion statement. The example is relatively simple, because the choice of the expression used to define a value for PAY(I) always depends on a single condition. A more complex situation occurs when the choice of defining expression depends on the truth value of two or more conditions. Extending the above example, the amount of pay within each job type could also vary depending on how long the person was with the company. To keep track of this we need a new variable called YEARS(I). When the number of years someone is with our company reaches 10, that person will get a raise.

The expanded assertion to calculate pay could be as follows:

```
PAY( I ) = IF JOB( I ) = 'PROGRAMMER-ANALYST'
         THEN IF YEARS( I ) < 10
              THEN 50.00*HOURS( I )
              ELSE 100.00*HOURS( I )
         ELSE IF JOB( I ) = 'JANITOR'
              THEN IF YEARS( I ) < 10
                   THEN 12.50*HOURS( I )
                   ELSE 15.00*HOURS( I )
              ELSE IF YEARS( I ) < 10
                   THEN 2.17*HOURS( I )
                   ELSE 2.18*HOURS( I );
```

This new example, shows the nesting of an additional condition and two defining expressions within each defining expression of the first example, which nested an additional condition and two defining expressions within the first ELSE expression. The choice of which expression is used to define the dependent variable depends on two conditions. (When writing your own nested conditional expressions, you'll find it helpful, as shown above, to indent, so that conditions and defining expressions at the same level of depth are vertically aligned.)

The syntax for the PL/1 form of conditional assertion statement does not allow you to nest conditions and expressions within the first defining expression, preceded by THEN. To make the choice of definition for the dependent variable depend on two conditions, you'll have to use the more cumbersome technique of writing compound Boolean expressions, with two comparison expressions connected by a logical and (as described in Chapter 4). The assertion defining values for PAY(I) would therefore be written as follows:

```
IF JOB(I) = 'PROGRAMMER-ANALYST' & YEARS(I) < 10
THEN PAY(I) = 50.00*HOURS(I);
ELSE IF JOB(I) = 'PROGRAMMER-ANALYST'
     THEN PAY(I) = 100.00*HOURS(I);
     ELSE IF JOB(I) = 'JANITOR' & YEARS(I) < 10
          THEN PAY(I) = 12.50*HOURS(I);
          ELSE IF JOB(I) = 'JANITOR'
               THEN PAY(I) = 15.00*HOURS(I);
               ELSE IF YEARS(I) < 10
                    THEN PAY(I) = 2.17*HOURS(I);
                    ELSE PAY(I) = 2.18*HOURS(I);
```

For this reason, the ALGOL-like syntax is the preferred form to use when writing nested conditional assertion statements.

CHAPTER 7

USING SUBSCRIPTS IN ASSERTION STATEMENTS

7.1 OVERVIEW

Whenever we mentioned subscripts earlier in this text, we were really talking about subscript expressions. Subscript expressions specify which elements of a subscripted variable are to be used in an assertion statement as independent or dependent variables. (Subscripted variables are defined in Section 6.2.) We express a subscripted variable by following the name of a repeating data structure (usually a FIELD) with one or more subscript expressions, separated by commas. The following is a sample of the variety of legal subscript expressions used in subscripted variables:

```
HAWKEYE(I,J)
BJ(I,J+3)
RADAR(KLINGER(K))
HOT_LIPS(FOR_EACH.BURNS)
```

Subscript expressions are arithmetic expressions, as defined in Section 4.9.3. These expressions give integer values corresponding to the postions of elements in a data array or tree, with one subscript expression for each dimension or subscript (see Section 2.2). (The word index is also used to stand for the value of a subscript expression.) If the values of the subscript expressions of a subscripted dependent variable are constants, then the definition taking place in the assertion will apply only to the element with those indices. For example, the assertion statement

```
ALBERT( 2 ) = 3;
```

defines the second element of the subscripted variable ALBERT as equal to 3, but does not affect the value of any of the other elements.

An assertion like the one above, which defines only one element of an array is very inefficient. When a subscripted variable contains a subscript expression that can take on a range of values, an assertion statement can simultaneously define values for all the elements of that array variable whose index values are in that range. In an assertion statement such as

```
BEAN( I ) = 3;
```

where the defining expression is a constant, all elements of the dependent variable BEAN(I) are given the same value. Alternatively, if both independent and dependent variables are subscripted, then each element of the dependent variable can be defined differently, depending on the value of the corresponding element of the independent variable. For example, in

```
WILMA( I ) = FRED( I );
```

each element of WILMA(I) is defined as equal to the element of FRED(I) with the same index value of I.

By including conditions, you can limit which elements of subscripted independent variables will be used to define the dependent variable. For example, in

```
BETTY( I ) = IF I < 4 THEN 0 ELSE BARNEY( I )*17;
```

only elements of BARNEY(I) with subscripts of 4 or greater will be used

in defining values of BETTY(I). If a variable in the defining expression is subscripted, but the dependent variable is not, then you must include conditions to define the dependent variable in terms of only one of the elements of the independent variable. (Otherwise, you violate MODEL assumptions.) An example would be,

```
ROCKY = IF I = 4 THEN BULLWINKLE(I);
```

Although BULLWINKLE(I) can contain many elements, only the fourth one will be used to define ROCKY. (Alternatively, you could have written

```
ROCKY =  BULLWINKLE(4);                                    )
```

The definition of all the elements of an array in a single assertion, is done through the use of subscript variables. A subscript variable is an arithmetic variable which can take on any integer value from one up to the total number of elements of a subscripted variable (along a particular dimension). MODEL allows the use of global and local subscript variables. A global subscript variable takes the same range in all the assertions in which it is used throughout the specification, while a local subscript variable may take on a different range in each assertion. Global and local subscript variables are discussed in more detail in Section 7.3. A subscript expression may contain a subscript variable by itself, or as part of a complex arithmetic expression containing constants, functions, and variables, which may themselves be subscripted. The next section will examine how the different types of subscript expressions are classified.

7.2 TYPES OF SUBSCRIPT EXPRESSIONS AND THEIR USES

Subscript expressions in MODEL can be categorized according to their form. The MODEL processor compiles some forms more efficiently than others, so that these are preferred. The types of subscript expressions are as follows:

1) I,

2) I-K, where K > 0,

3) none of the others (e.g., constants),

4) X(I),

5) X(I-C)-K, where C + K = 1,

6) X(I-C)-K, where C + K > 1,

The preceding description of types of subscripts uses the following nomenclature:

I is a subscript variable which can take on any integer value in the range of the variable for which it is an index.

C and K are integer constants.

X is a sublinear indirect indexing vector (see below).

The MODEL compiler generates a more efficient program when you use Type 1 or 2 subscript expressions, then when you use Type 3. An indirect indexing vector is a subscripted variable which is used in the subscript expression of another subscripted variable. The use of indirect indexing vectors as subscript expressions is optimized when those indirect indexing vectors are sublinear, as illustrated by Type 4, 5, and 6 subscript expressions above. X(I) is a sublinear indirect indexing vector if it is defined by an assertion of the form:

```
X(I) = IF I = 1
       THEN 0 ¦ 1
       ELSE IF ‹Boolean Expression›
            THEN X(I-1)
            ELSE X(I-1) + 1;
```

In words, this says that X(I) is equal to either 1 or 0, when I is equal to 1. When I is greater than 1, the value of X(I) for each I, is defined as equal to either the value of X(I-1) or X(I-1) plus 1, depending on a condition. The effect is that the sublinear indirect indexing vector X(I) always takes integer values and is monotonically

increasing with I. It is also less than or equal to I, for any I, because I always increases by 1 (linearly), while X(I) increases by 0 or 1 (less than linearly or sublinearly).

The usefulness of sublinear indirect indexing vectors in business applications is illustrated by the following specification, displayed in Figure 7.1. We start with a FILE of life insurance information. The FILE is called DATA. Each RECORD in the FILE contains three FIELDS: the first to give the year of birth of the policy-holder, the second to give his or her name, and the third to give the amount he or she pays each year. The three FIELDS are called BIRTH, NAME, and PREMIUM, respectively. We decide to separate the RECORDS into three distinct TARGET FILES based on year of birth, so that we can be sure everyone is paying the appropriate premiums. (These TARGET FILES will have the same data structure, except for number of RECORDS, as the original SOURCE FILE.) We want the RECORDS of everyone born before 1920 to go into the first TARGET FILE, OLDER, the RECORDS of people born from 1920 to 1950 to go into the second FILE, MIDDLE, and the RECORDS of people born after 1950 to go into the third, YOUNGER.

```
MODULE: INSURANCE;
SOURCE: DATA;
TARGET: OLDER,MIDDLE,YOUNGER;


(DATA, OLDER, MIDDLE, YOUNGER) ARE FILES (R(*));
     R IS RECORD (BIRTH,NAME,PREMIUM);
          BIRTH IS FIELD (PIC '9999');
          NAME IS FIELD (CHAR (20));
          PREMIUM IS 20 (PIC 'ZZ99');


INT IS FILE (DATE1 (*), DATE2 (*), DATE3 (*));
     DATE1 IS FIELD (NUM (5));
     DATE2 IS FIELD (NUM (5));
     DATE3 IS FIELD (NUM (5));


I IS SUBSCRIPT;


DATE1(I) = IF DATA.BIRTH(I) < 1920
           THEN IF I = 1
                THEN 1
                ELSE DATE1(I-1) + 1
           ELSE IF I = 1
                THEN 0
                ELSE DATE1(I-1);


DATE2(I) = IF DATA.BIRTH(I) >= 1920 & DATA.BIRTH(I) <= 1950
           THEN IF I = 1
                THEN 1
                ELSE DATE2(I-1) + 1
           ELSE IF I = 1
                THEN 0
                ELSE DATE2(I-1);


DATE3(I) = IF DATA.BIRTH(I) > 1950
           THEN IF I = 1
                THEN 1
                ELSE DATE1(I-1) + 1
           ELSE IF I = 1
                THEN 0
                ELSE DATE3(I-1);
```

(FIGURE 7.1 CONTINUED NEXT PAGE)

```
OLDER.BIRTH(DATE1(I)) = IF I = 1 & DATE1(I) = 1
                        THEN DATA.BIRTH(I)
                        ELSE IF DATE1(I) > DATE1(I-1)
                             THEN DATA.BIRTH(I);

OLDER.NAME(DATE1(I)) = IF I = 1 & DATE1(I) = 1 THEN DATA.NAME(I)
                       ELSE IF DATE1(I) > DATE1(I-1) THEN DATA.NAME(I)

OLDER.PREMIUM(DATE1(I)) = IF I = 1 & DATE1(I) = 1 THEN DATA.PREMIUM(I)
                          ELSE IF DATE1(I) > DATE1(I-1) THEN DATA.PREMIUM(I);

MIDDLE.BIRTH(DATE2(I)) = IF I = 1 & DATE2(I) = 1 THEN DATA.BIRTH(I)
                         ELSE IF DATE2(I) > DATE2(I-1) THEN DATA.BIRTH(I);

MIDDLE.NAME(DATE2(I)) = IF I = 1 & DATE2(I) = 1 THEN DATA.NAME(I)
                        ELSE IF DATE2(I) > DATE2(I-1) THEN DATA.NAME(I);

MIDDLE.PREMIUM(DATE2(I)) = IF I = 1 & DATE2(I) = 1 THEN DATA.PREMIUM(I)
                           ELSE IF DATE2(I) > DATE2(I-1) THEN DATA.PREMIUM(I);

YOUNGER.BIRTH(DATE3(I)) = IF I = 1 & DATE3(I) = 1 THEN DATA.BIRTH(I)
                          ELSE IF DATE3(I) > DATE3(I-1) THEN DATA.BIRTH(I);

YOUNGER.NAME(DATE3(I)) = IF I = 1 & DATE3(I) = 1 THEN DATA.NAME(I)
                         ELSE IF DATE3(I) > DATE3(I-1) THEN DATA.NAME(I);

YOUNGER.PREMIUM(DATE3(I)) = IF I = 1 & DATE3(I) = 1 THEN DATA.PREMIUM(I)
                            ELSE IF DATE3(I) > DATE3(I-1) THEN DATA.PREMIUM(I);
```

Figure 7.1

Sample Specification Using Sublinear Indirect Indexing Vectors

We can accomplish this task using three sublinear indirect indexing vectors: DATE1(I), DATE2(I), and DATE3(I), to keep track of the FIELDS to be placed into the three TARGET FILES. (Figure 7.2 shows sample values of the three sublinear indexing vectors for different values of I and DATA.BIRTH(I).) We define DATE1(I) as follows:

```
DATE1(I) = IF DATA.BIRTH(I) < 1920
     THEN IF I = 1
          THEN 1
          ELSE DATE1(I-1) + 1
     ELSE IF I = 1
          THEN 0
          ELSE DATE1(I-1);
```

so that it only increases when I corresponds to a RECORD containing a BIRTH FIELD with a value less than 1920. We can then write the assertion

```
OLDER.BIRTH(DATE1(I)) = IF I = 1 & DATE1(I) = 1
                        THEN DATA.BIRTH(I)
                        ELSE IF DATE1(I) > DATE1(I-1)
                             THEN DATA.BIRTH(I);
```

using DATE1(I) as a subscript expression, to keep track of the BIRTH FIELDS entered into the TARGET FILE OLDER. When DATA.BIRTH(I) has a value greater than 1920, and DATE1(I) doesn't change, then nothing is added to the new FILE. However, every time DATE1(I) is increased by 1, the FIELD BIRTH (which has the index I in the FILE DATA and the index DATA1(I) in FILE OLDER) is added to TARGET FILE OLDER.

I	DATA.BIRTH(I)	DATE1(I)	DATE2(I)	DATE3(I)
1	1915	1	0	0
2	1930	1	1	0
3	1947	1	2	0
4	1953	1	2	1
5	1918	2	2	1
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Figure 7.2

Sample Values for Several Sublinear Indirect Indexing Vectors

72;.b 2

DATE1(I) is used in the same way to keep track of the of the FIELDS DATA.NAME(I) and DATA.PREMIUM(I). We write parallel assertions to add them to the FILE OLDER when the value of DATE1(I) increases. Similarly, we can define the sublinear indirect indexing vectors DATE2(I) and DATE3(I) as increasing for values of DATA.BIRTH(I) in the ranges 1920 to 1950 and greater than 1950, respectively. This allows us to write assertions defining the values of MIDDLE.BIRTH(DATE2(I)), MIDDLE.NAME(DATE2(I)), MIDDLE.PREMIUM(DATE2(I)), YOUNGER.BIRTH(DATE3(I)), YOUNGER.NAME(DATE3(I)), and YOUNGER.PREMIUM(DATE3(I)).

7.3 SUBSCRIPT VARIABLES

Subscript variables in MODEL fall into two classes, global and local, which differ in their scope of applicability throughout the specification. The two types of subscript variables are defined syntactically in Figure 7.3.

```
1  ‹subscript variable› ::= ‹global subscript variable› ¦
                            ‹local subscript variable›
   2  ‹global subscript variable› ::= ‹name› ¦ FOR_EACH.‹name›
   2  ‹local subscript variable› ::= SUB1 ¦ SUB2 ¦ SUB3 ¦ SUB4 ¦ SUB5 ¦
                                     SUB6 ¦ SUB7 ¦ SUB8 ¦ SUB9 ¦ SUB10
```

Figure 7.3

Types of Subscript Variables

A global subscript has the same range in all the assertion statements in which it is used. For example, the global subscript variable I, once it has been declared in a statement like

```
I IS SUBSCRIPT (10);
```

can be used in different assertions as part of the subscript expressions of different variables, and it will always take the same range (see example below).

```
CHESTER(I) = JESSICA(I) - DONAHUE(I);
CORRINE(I) = EUNICE(11-I);
```

There are two types of global subscript variables. The first is declared in a subscript declaration statement. The second is a qualified name variable of the form FOR_EACH.X, where X is the name of a data structure. The syntax for declaring global subscript variables in a subscript declaration statement is shown in Figure 7.4. The following are examples of legal subscript declaration statements:

```
ROBIN IS SUBSCRIPT;
(BATMAN,SUPERMAN) ARE SUBSCRIPTS (10);
FLASH SUB (15);
```

The examples show that more than one global subscript variable can be declared in the same statement (based on the definition of <variable> in Figure 5.4), with each one having the same range. Also, declaring ranges for global subscript variables is optional in certain situations (see below).

```
1  <subscript declaration statement> ::=
   <variable> [<IS>] <SUBSCRIPT> [(<range of subscripts>)];
   2  <SUBSCRIPT> ::= SUBSCRIPT|SUBSCRIPTS|SUB|SUBS
   2  <range of subscripts> ::= <unsigned integer>
```

Figure 7.4

Syntax for Global Subscript Declaration

Global subscript variables do not always need to be declared, and if declared, their ranges do not always need to be specified in subscript declaration. The MODEL compiler can sometimes calculate a range for a global or local subscript variable which was not declared, as long as the variable is used as part of a subscript expression in an assertion statement. Range propagation is the process of assigning ranges (or repetition counts) to global and local subscript variables. This can be done because each specification contains several sources of information about ranges. For example, if the range of a subscript variable is not declared, but the subscript variable is used with a SOURCE FILE or TARGET FILE variable with a declared range, then the subscript variable will automatically take this range in all other assertions in which it appears. Once the range of a subscript variable is set, then the MODEL compiler will know this to be the range of every other variable using it as a subscript, even ones whose ranges weren't specified. Assertions that define control variables, prefixed with SIZE or END, and the positions of ENDFILE markers are additional sources of information used in range propagation as well (see Sections 8.2 and

8.3).

Another way to create a global subscript variable is to add the prefix FOR_EACH to the name of a data structure. A qualified name global subscript variable, such as FOR_EACH.KIRK, will have the same range as the rightmost subscript dimension of the variable, KIRK, whose name it incorporates. (Hence a FOR_EACH global subscript is one-dimensional.) Once a qualified subscript is defined, it can be used as a subscript for other variables with the same range. If SPOCK had the same range as KIRK in its rightmost subscript dimension, then SPOCK(FOR_EACH.SPOCK) and SPOCK(FOR_EACH.KIRK) would be equivalent.

Predefined local susbscript variables named SUB1, SUB2, ..., SUB10 are available in the MODEL system. Each of these local subscript variables may have a different range in each assertion in which it is used. Note that if the same local subscript variable is used in the subscript expressions of several variables in the same assertion, then it will have the same range in all of those variables. The ranges of local subscript variables are not declared, but instead are derived by the MODEL compiler through range propagation.

7.4 CONVENTIONS FOR SUBSCRIPT OMISSION

As explained previously, variables in MODEL assertion statements are always FIELD variables. For a repeating FIELD variable of n dimensions, it takes n subscript expressions to distinguish each individual element. However in an assertion statement containing several subscripted variables, copying long lists of subscripts after each variable can get quite tedious. Therefore MODEL has a convention for omitting subscripts. Subscript expressions to be omitted must have the following characteristics:

1) They are common to all variables in an assertion.
2) They are Type 1 subscript expressions (as defined in Section 7.2).
3) They are not used as independent variables in the assertion, as in:

```
SHEEBA(I,J) = 4 * I + J;
```

4) They are in the same order in each variable from which they are removed.
5) They are on the left of other subscript expressions (or have no subscript expressions to the right).

If all the subscript expressions in an assertion have the above characteristics, then they can be omitted, without changing the meaning of the assertion, as in,

```
A(I,J,K) = 2 * B(I,J,K) + C(I,J);
A(J,K) = 2 * B(J,K) + C(J);
A(K) = 2 * B(K) + C;
```

CHAPTER 8

CONTROL VARIABLES

8.1 OVERVIEW

This Chapter will explain how to use control variables in MODEL assertion statements. By the end of this Chapter you should know everything you will need about how to write a MODEL specification. Control variables allow you to use assertion statements to define data attributes that you chose not to specify in data declaration, such as the range of a subscripted variable, the length of a piece of data, or the POINTERS for the RECORDS of an ISAM FILE. When you define a control variable, you can use it as an independent part of a defining expression or condition in another assertion. In these ways, control variables increase your flexibility in setting up your specifications.

Control variables are qualified name variables, which means they are constructed by attaching MODEL keyword prefixes to variable names. (The prefix is attached to the variable name by using a period, as in SIZE.CHARLES; see Section 4.6 for more information.) Although the keyword prefixes of control variables may be attached to the names of GROUPS or RECORDS (as in POINTER.X), the control variables themselves act as FIELDS, each holding an individual piece of data. The MODEL control variable keyword prefixes to be discussed in this chapter are SIZE, END, LENGTH, NEXT, SUBSET, POINTER, and FOUND.

8.2 SIZE.X

SIZE is one of two keyword prefixes in the MODEL language used to define the range of some dimension of an array variable, if that range was not specified in data declaration. The other prefix used for this

purpose is END, discussed in the next section. If X is a subscripted FIELD, GROUP, or RECORD variable, then SIZE.X may be used to define and represent the number of elements in the rightmost subscript dimension of X in terms of an arithmetic expression whose value is an integer. (SIZE.X may be defined equal to 0; see Section 5.3.3 on optional data structures.) For example, suppose X was declared as follows:

```
3  X (*) IS FIELD (PIC '99');
```

A later assertion of the form

```
SIZE.X = 10;
```

will set the range of X to 10 in any assertions in which X is used with a global subscript, for example

```
X(I) = IF I = 1 THEN 1 ELSE I**2 - 1;
```

(If the range of I was declared in a subscript declaration statement, then an assertion defining SIZE.X is unnecessary, because X(I) is given the same range as I through range propagation, as explained in Section 7.3.)

If X has more than one subscript, then SIZE.X may have subscripts as well, although the total number will always be at least one less than in X. This is because we can define the number of elements in a one-dimensional vector with a zero-dimensional scalar, as the above example shows. If SIZE.X is subscripted, then each of its elements will be a scalar, giving the range of a vector in X (see below).

If variable X has n subscripts, we can define SIZE.X variables having from 0 to n-1 subscripts, as illustrated in the following example. Consider FILE Z, declared in Figure 8.1(a), which contains

RECORDS Y and FIELDS X. Y and X repeat an unknown number of times. Subscript I is used to distinguish the elements of Y, and subscript J distinguishes the elements of X. Figures 8.1(b) and 8.1(c) display two possible data arrays which are consistent with this declaration. In the first, each RECORD contains the same number of FIELDS, while in the second, the number of FIELDS vary between RECORDS. Our goal is to write assertions to define the ranges of X and Y for each of the two arrays.

```
1  Z IS FILE,
     2  Y(*) IS RECORD,
          3  X(*) IS FIELD (PIC '9');
I IS SUBSCRIPT;
J IS SUBSCRIPT;
```

(a) Declaration of FILE Z

	J		
I	1	2	3
1	4	9	7
2	3	6	1

(b) Possible Structure for X(I,J)

	J		
I	1	2	3
1	2	8	3
2	5	7	
3	6	11	9

(c) Another Structure for X(I,J)

Figure 8.1

Data Structures for FILE Z

RECORD variable Y(I) is one-dimensional, so that its range can be defined with a single scalar in a simple assertion statement such as:

SIZE.Y = 2; for the first array (Figure 8.1(b)), and

SIZE.Y = 3; for the second (Figure 8.1(c)).

FIELD variable X(I,J) is two-dimensional. In the first array, the range of X(I,J) is the same no matter what the value of I. In this situation, we can write a simple assertion to define one range for X(J) which holds for the whole FILE, as in

```
SIZE.X = 3;
```

In the second array, the range of X(I,J) changes for each RECORD, as the value of subscript I changes. The range of X(J) can no longer be defined in a simple assertion. Instead a conditional assertion is required, for example

```
SIZE.X(I) = IF I = 2 THEN 2 ELSE 3;
```

In this case, a vector of scalars is needed to define the range of X(I,J), because the range of J depends on the value of I.

We saw in the above example, that we could completely define the range of the doubly subscripted X(I,J) with either the singly subscripted SIZE.X(I) or the non-subscripted SIZE.X, depending on the structure of X(I,J). When we removed subscripts from X(I,J) to form SIZE.X(I) and SIZE.X, we removed the rightmost subscript first. In SIZE.X(I), the subscript remaining has the same range as the comparable subscript in the original repeating variable X(I,J). These principles generalize to the writing of assertions using SIZE qualified control variables to define the range of data arrays containing many more dimensions and susbscripts. Also, if SIZE.X has m subscripts and X has n subscripts, where 0 <= m < n, then the value of SIZE.X cannot be a function of any subscript s, where m < s <= n. In other words, any

subscripts which have been been removed from X in forming SIZE.X, cannot be used in defining SIZE.X.

8.3 END.X

The MODEL keyword prefix END provides an alternative to SIZE in forming a control variable to define ranges in assertion statements. Unlike SIZE.X, END.X is a Boolean variable (defined in Section 4.8), and takes the same number of subscripts as repeating variable X. Each element of END.X consists of a single bit value of true or false. An element of END.X (distinguished by n supscripts) is defined as true when the rightmost subscript of X takes its maximum value. Otherwise, each element of END.X is defined as false.

The array END.X is usually defined in terms of a comparison expression which will be true only when the rightmost subscript takes its maximum range. To define the range of a vector S(I) as equal to some constant K, we could write

```
END.S(I) = I = K;
```

END.S(I) would be true when I equals K and false otherwise, thereby defining the range of S(I) as K.

IF X contains two or more dimensions, then the array END.X can be used to set the ranges of several vectors at the same time. For example, to define the range of variable X(I,J) from Figure 8.1(b), we could write the simple assertion

```
END.X(I,J) = J = 3;
```

thereby defining the range of X(I,J) as 3, irrespective of the value of I. This effectively defines the range of two vectors at the same time, X(I,J) for I equalling 1 and X(I,J) for I equalling 2. The values of the Boolean elements of END.X(I,J) based on the above assertion are

shown in Figure 8.2(a).

	J		
I	1	2	3
1	F	F	T
2	F	F	T

(a) Possible Values of Elements of END.X(I,J)

	J		
I	1	2	3
1	F	F	T
2	F	T	
3	F	F	T

(b) Alternative Values for Elements of END.X(I,J)

Figure 8.2

Values of Elements of END.X(I,J)

However, sometimes the range of a variable in a particular dimension may change depending on the value of a subscript for a higher dimension in the array. This is the case for the range of X(I,J) as illustrated in Figure 8.1(c). To define the range of the three vectors X(I,J) for I equalling 1, 2, and 3 we need to write a conditional assertion statement in which the value of END.X(I,J) depends on both the value of I and the value of J. Such an assertion would be

```
END.X(I,J) = IF I = 2 THEN J = 2 ELSE J = 3;
```

The values of the elements of END.X(I,J) defined by this assertion are shown in Figure 8.2(b). END.X(I,J) is defined as true when J equals 3

for the first vector, when J equals 2 for the second, and when J equals 3 for the third.

As an alternative, you can make range definition via an END qualified control variable depend on the value of the elements of the original variable, rather than their subscripts. You do this by setting up the data so that the last element of the subscripted variable with an unspecified range is given a unique value, such as 999, making it a termination marker. You can then write an assertion like

```
END.Z(I) = Z(I) = 999;
```

This will define END.Z(I) as true when Z(I) equals the termination value of 999.

You can also use the comparison expression containing an END qualified control variable as the Boolean expression in the condition in a conditional assertion statement. This way you can make the value that a dependent variable is assigned depend on whether the END control variable is true. For example, we could write

```
A(I) = IF END.Y(I) THEN 6 ELSE 8;
```

This defines the value of A(I) as 6 when END.Y(I) is true (at the maximum range of Y(I)), and 8 otherwise.

Sometimes, defining unspecified variable ranges through SIZE or END qualified control variables is not necessary, even if they cannot be inferred from subscripts through range propagation. This occurs when variable ranges may be obtained by the MODEL compiler from an ENDFILE marker. Whenever a set of data is read from a data FILE, the last element of that FILE is automatically marked. This last element can be used by the compiler to define the range of a subscripted varible, just as you would define it using an END control variable. This repeating variable can be a RECORD or GROUP. The restrictions are that there can be only one subscripted variable (with uncertain range) in the FILE, and

that the last element of this repeating variable has to occur at the end of the FILE. When these restrictions are satisfied, the unspecified range of the subscripted variable will be defined automatically. Otherwise, range definition through SIZE or END control variables are required.

8.4 LEN.X

The LEN.X control variable is used to define the length (number of characters) in character FIELD variable X, when this information is left unspecified in data declaration. (As described in Section 5.6.1, the length of a FIELD variable of character data type may be declared as a minimum-maximum range.) If X is subscripted, then LEN.X will have the same number and range of subscripts, for example

```
LEN.MARIAN = 7;
LEN.EVE(I) = IF I < 5 THEN 9 ELSE 12;
```

Each element in LEN.X(I) is a scalar defining the length of the element of X(I) with the corresponding subscript in terms of any arithmetic expression whose value is an integer. The only restriction is that the value of LEN.X cannot depend on the value of any FIELDS physically positioned after FIELD(S) X in the same SOURCE FILE RECORD.

8.5 MALDATA.X

MALDATA.X is a Boolean variable which takes the same number of subscripts as repeating SOURCE RECORD X. If a conversion error occurred when reading in a FIELD of X, then the element of MALDATA.X corresponding to that RECORD is defined as true. Otherwise, each element of MALDATA.X is defined as false.

8.6 NEXT.X

If X is a FIELD in a RECORD, then NEXT.X has a value equal to the contents of the corresponding FIELD in the same position in the following RECORD in the FILE. (This will be the RECORD with the next higher RECORD-level subscript, unless it is the last RECORD in a GROUP, in which case the next RECORD will be the first one in the next GROUP.) NEXT.X can be of any data type, depending on the data type of X.

We can use NEXT.X to define the range of a GROUP of RECORDS by making the definition of the END of the GROUP contingent on the Boolean value of an expression comparing X and NEXT.X. For example, Figure 8.3 shows the data declaration of a FILE (named SALES) where RECORDS (named INVOICE) are placed in GROUPS (named PRODUCT), but the number of RECORDS in each GROUP is unspecified. Each RECORD contains an identifying FIELD called PIN, short for product identification number, which has a common value for all the RECORDS in any one GROUP, but is different among GROUPS. By writing an assertion like

```
END.INVOICE(I,J) = PIN(I,J) ^= NEXT.PIN(I,J);
```

we can define the Boolean variable END.INVOICE(I,J) as true, specifying that a particular RECORD is the last one in a PRODUCT GROUP, when the value of PIN is different in the following RECORD. This allows the number of INVOICE RECORDS in each PRODUCT GROUP to be determined.

```
1 SALES IS FILE,
  2 PRODUCT(41) IS GROUP,
    3 INVOICE(*) IS RECORD,
      4 PIN IS FIELD (PIC '999999'),
      4 QUANT IS FIELD (PIC '9999'),
      4 PRICE IS FIELD (PIC '999V.99');
I IS SUBSCRIPT;
J IS SUBSCRIPT;
```

Figure 8.3

Data Declaration of FILE SALES

NEXT.X cannot be used for RECORDS in ISAM FILES. A second restriction is that the number of FIELDS to the left of FIELD X should be fixed in the RECORD. If there are a varying number of FIELDS to the left of FIELD X in each RECORD, then the NEXT.X FIELD will not be located correctly. There is no problem, however, if there is a varying number of FIELDS to the right of X in each RECORD.

8.7 SUBSET.X

SUBSET.X is a Boolean control variable in which each element corresponds to an element of subscripted RECORD variable X. You can define each element of SUBSET.X as true or false, depending on whether you want the RECORD from X with the corresponding subscripts to be included in a TARGET FILE. For example, to omit the second of the three RECORDS in TARGET FILE E, declared below in Figure 8.4, you could write

```
SUBSET.F(I) = I ^= 2;
```

The above assertion defines a value of true for all the elements of SUBSET.F(I), except the second. Those RECORDS of F(I) for which SUBSET.F(I) is true will be included in TARGET FILE E; the second RECORD, for which SUBSET.F(I) is false, won't be.

```
1 E IS FILE,
  2 F(3) IS RECORD,
    3  OUT IS FIELD (PIC '9');
I IS SUBSCRIPT;
```

Figure 8.4

TARGET FILE to Demonstrate the Use of SUBSET

The use of SUBSET does not affect computations, only what is written at the end as TARGET data. For example, you could define a FIELD in another TARGET FILE in terms of omitted FIELD OUT(2). Therefore you must be sure to declare the full range of RECORDS for your TARGET FILE, including those to be omitted.

8.8 POINTER.X

The next two keywords, POINTER and FOUND, are used with keyed FILES. Section 5.4 describes how each RECORD, X, in a keyed FILE has a FIELD, called a KEY, which contains a unique identifying alphanumeric string. The control variable POINTER.X contains an array of these strings, called POINTERS, as elements. When you define the strings making up POINTER.X, you simultaneously define the desired organization of RECORDS of the keyed FILE. The keyed FILE is rearranged, in terms of number and ranges of dimensions, so that the positions of the RECORDS, as identified by their KEYS, match up with the positions of the corresponding POINTERS in POINTER.X. For example, the assertion statement

```
POINTER.X( I ) = Y( I );
```

would shape the RECORDS of the keyed FILE into a one-dimensional vector, while the statement

```
POINTER.X( I,J ) = Z( I,J );
```

would reorder them into a two-dimensional matrix. RECORDS from the keyed FILE having KEYS for which there are no corresponding POINTERS will be excluded from the reorganized FILE.

POINTER.X is usually defined in terms of an array of FIELDS taken from a separate reference SOURCE FILE. Once the RECORDS of the keyed FILE are given the structure of this array, you can use subscripts to refer to specific FIELDS from keyed RECORDS as independent or dependent variables in your assertion statements. In this way, you can use a keyed FILE as a SOURCE FILE, a TARGET FILE, or as both a SOURCE and TARGET FILE. In the last case, as explained below, you can use POINTER and FOUND to easily update certain RECORDS in a keyed FILE while leaving others unchanged.

For example, Figure 8.5 illustrates the specification you would write to define the FIELDS of a TARGET FILE in terms of the FIELDS from a keyed SOURCE FILE. SOURCE FILE E is declared as ISAM, with single RECORD variable FT containing KEY FIELD OUT1 and additional FIELD OUT2. The contents of SOURCE FILE E are shown in Figure 8.6(a). In the reference SOURCE FILE, B, we declare a two-dimensional FIELD variable, D(I,J), which contains the POINTERS we will use to restructure FILE E. The elements of D(I,J) are shown in Figure 8.6(b). We then use the statement

```
POINTER.FT(I,J) = D(I,J);
```

to define the shape of keyed RECORDS as the two-dimensional matrix. The FIELD OUT2 of the keyed RECORD variable FT can then be described in terms of this structure, allowing the FIELD variable K(I,J) in TARGET FILE G to be defined. The values of the elements of TARGET FIELD K(I,J), defined from SOURCE FIELD OUT2(I,J), are shown in Figure 8.6(c).

```
MODULE: A;
SOURCE: B,E;
TARGET: G;

1 B IS FILE,
  2 C(*) IS RECORD,
    3 D(*) IS FIELD (PIC '9');

1 E IS FILE KEY IS OUT1 ORG IS ISAM,
    2 FT IS RECORD,
    3 OUT1 IS FIELD (PIC '9'),
    3 OUT2 IS FIELD (PIC 'Z99');

1 G IS FILE,
  2 H(*) IS RECORD,
    3 K(*) IS FIELD (PIC 'Z99');

I IS SUBSCRIPT (2);
J IS SUBSCRIPT (3);

POINTER.FT(I,J) = D(I,J);

K(I,J) = OUT2(I,J);
```

Figure 8.5

Example Using POINTER with Keyed FILE as SOURCE

If the POINTER variable contains an identifying string which is not a KEY for any RECORD in the keyed FILE, then the message

RECORD NOT FOUND IN FILE filename WITH KEY key-value

is printed. Also, if the KEY of a particular RECORD from the keyed FILE is not included in the POINTERS from the reference FILE, then the FIELDS of that RECORD will not be used to define the values of FIELDS in the TARGET FILE. For example, elements of SOURCE FIELD OUT2 from RECORDS of ISAM FILE E with KEY values of 7, 8, and 9, as shown in Figure 8.6(a), will not be used to define TARGET FIELD K(I,J).

RECORD FT

KEY FIELD OUT1	FIELD OUT2
6	19
3	27
4	33
5	64
7	95
8	128
9	43
1	29
2	203

(a) Contents of ISAM SOURCE FILE E

(Figure 8.6 Continued Next Page)

I	J 1	2	3
1	2	4	6
2	5	3	1

(b) Contents of D(I,J)

I	J 1	2	3
1	203	33	19
2	64	27	29

(c) Contents of K(I,J)

Figure 8.6

Contents of Data Structures in Specifcation of Figure 8.5

Using a keyed FILE as TARGET is an easy way to fill an empty SAM or ISAM FILE with data. For example, Figure 8.7 shows the specification you would write to define the FIELDS of keyed ISAM TARGET FILE M. (Here the RECORDS of the keyed FILE will be shaped into a vector rather than a matrix.) Whenever the POINTER gives a KEY that is not already contained in the keyed TARGET FILE, then a new RECORD for that FILE is defined, with its KEY equal to the value of the POINTER FIELD. (This method could also be used to add RECORDS to an already existing keyed FILE.) When the POINTER gives a KEY value which corresponds to a RECORD already contained in the TARGET FILE, nothing is changed and the message

FILE filename IS TARGET ONLY, RECORD UPDATE WITH KEY key-value IS IGNORED

is printed.

```
MODULE: A;
SOURCE: B;
TARGET: M;

1 B IS FILE,
  2 C(*) IS RECORD,
    3 D IS FIELD (PIC '9'),
    3 G IS FIELD (CHAR (5));

1 M IS FILE KEY IS OUT1 ORG IS ISAM,
  2 F IS RECORD,
    3 OUT1 IS FIELD (PIC '9'),
    3 OUT2 IS FIELD (CHAR (5));

I IS SUBSCRIPT;

POINTER.FT(I) = D(I);

OUT2(I) = G(I);
OUT1(I) = D(I);
```

Figure 8.7

Example Using POINTER with Keyed FILE as TARGET

The most common usage of the POINTER control variable is with a keyed FILE which is used as both SOURCE and TARGET. This allows you to alter certain RECORDS in the keyed FILE while not changing others. For example, you could update the RECORDS of those items in a stock inventory which just arrived in a shipment, while not affecting the RECORDS of your other items. Figure 8.8 shows the ISAM FILE E used as both a SOURCE FILE and a TARGET FILE. This example is basically similar to the two previous ones (see also the example in Section 5.4). What is different is that every time one of the variables from the ISAM FILE is mentioned in an assertion, it must be accompanied by a keyword prefix NEW or OLD to tell if it refers to the variable before or after the ISAM FILE was updated. (See Section 4.6 for a discussion of the use of the keywords NEW and OLD.) RECORDS with KEYS not included among the list of POINTERS will not be updated before being entered into the TARGET FILE. Unless redefined in an additional assertion, the KEY FIELDS will also be the same in both the SOURCE and TARGET versions of the keyed FILE.

```
MODULE: A;
SOURCE: B,E;
TARGET: E;

1 B IS FILE,
  2 C(*) IS RECORD,
    3 D IS FIELD (PIC '9'),
    3 G IS FIELD (PIC 'Z99');

1 E IS FILE KEY IS OUT1 ORG IS ISAM,
  2 FT IS RECORD,
    3 OUT1 IS FIELD (PIC '9'),
    3 OUT2 IS FIELD (PIC 'Z99');

I IS SUBSCRIPT;

POINTER.OLD.FT(I) = D(I);

NEW.OUT2(I) = OLD.OUT2(I) + G(I);
```

Figure 8.8

Example with POINTER Using Keyed FILE as both SOURCE and TARGET

8.9 FOUND.X

FOUND.X is a Boolean variable with the same size and shape as POINTER.X, that is, with the same range and number of dimensions. It is also used with keyed FILES, which usually have INDEX SEQUENTIAL organization. A element of FOUND.X contains a value of true, if the corresponding RECORD, whose KEY is given in POINTER.X, actually exists in the keyed FILE. If the RECORD does not exist, then the value of the element of FOUND.X with corresponding subscripts is false.

An example of the use of FOUND.X is the following assertion which can be added to the specification in Figure 8.7. The assertion is

```
SUBSET.NEW.FT = FOUND.OLD.FT;
```

and it uses both FOUND and SUBSET control variables. The effect of the assertion is to delete from the updated ISAM FILE all RECORDS which weren't changed (whose KEYS weren't pointed to and found). Only old RECORDS which were included in the update will remain in the keyed TARGET FILE.

APPENDIX A

FUNCTION LIST

A.1 SELECTED BUILT-IN PL/1 FUNCTIONS

ABS(x) — Arithmetic

ABS returns the absolute value of a given expression x. (it is the positive value of x)

BIT(x1[,x2]) — String-handling

Bit returns a bit string representation of a given value x1.

x1 expression to be converted.

x2 an expression that can be converted to integer specifying the length of the resulting bit string. If necessary, x2 is converted to a binary integer of precision (15,0). If x2 is omitted, the length is determined by the rules for type conversion.

CEIL(x) — Arithmetic

CEIL returns the smallest integer greater than or equal to a given value x.

If x is fixed-point with precision (p,q), the precision of the result is given by:

```
(MIN(N,MAX(P-Q+1,1)),0)
```

where N is the maximum number of digits allowable.

CHAR(x1[,x2]) — String-handling

CHAR returns a character string representation of a given value x1.

x1 expression to be converted.

x2 an expression that can be converted to integer specifying the length of the resulting character string. If necessary, x2 is converted to a binary integer of precision (15,0). If x2 is omitted, the length is determined by the rules for type conversion.

COPY(x1,x2)

The COPY built-in function copies a given string x1 a x2 specified number of times and concatenates the result into a single string.

x1 Any bit- or character-string expression. If the expression is a bit string, the result is a bit string. Otherwise, the result is a character string.

x2 Any expression that yields a nonnegative integer. The specified count controls the number of copies of the string that are concatenated, as follows:

Value of Count	String Returned
0	a null string
1	the string argument
n	concatenated copies of the string argument

Example

The function reference

```
COPY('12',3)
```

returns the character-string value '121212'.

COS(x)

The COS function returns a floating-point value that is the cosine of an arithmetic expression x, where x represents an angle in radians. The cosine is computed in floating point.

DATE

DATE returns a character string of length six, in the form yymmdd, where:

yy the current year
mm the current month
dd the current day

DECIMAL(x1[,x2[,x2]]) Arithmetic

Abbreviation: DEC(x1[,x2[,x3]])

DECIMAL returns the decimal representation of a given value x1 with a precision specified by x2 and x3.

x1 value to be converted to decimal base.

x2 unsigned decimal integer constant specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

x3 decimal integer constant, optionally signed, specifying the scale factor of the result. For a fixed-point result, if x2 is given and x3 is omitted, a scale factor of zero is assumed. For a floating-point result, only x2 can be given.

If both x2 and x3 are omitted, the precision of the result is determined from the rules for base conversion.

EXP(x)

The EXP built-in function returns a floating-point value that is the base e to the power of an arithmetic expression x. The computation is performed in floating point.

FIXED(x1[,x2[,x3]]) Arithmetic

FIXED returns the fixed-point representation of a given value x1 with a precision specified by x2 and x3.

x1 value to be converted to fixed-point scale.

x2 unsigned decimal integer constant specifying the total number of digits in the result.

x3 decimal integer constant, optionally signed, specifying the scale factor of the result. If x3 is omitted, a scale factor of zero is assumed.

If both x2 and x3 are omitted, the default value (15,0), for a binary result, or (5,0), for a decimal result, is assumed.

FLOAT(x1[,x2]) Arithmetic

FLOAT returns the floating-point representation of a given value x1 with

a precision specified by x2.

x1 value to be converted to floating-point scale.

x2 unsigned decimal integer constant specifying the total number of digits in the result. If x2 is omitted, the default value 21, for a binary result, or 6, for a decimal result, is assumed.

FLOOR(x) Arithmetic

FLOOR returns the largest integer less than or equal to a given value x.

If x is fixed-point with precision (p,q), the precision of the result is given by:

(MIN(N,MAX(p-q+1,1),0)

where N is the maximum number of digits allowable.

HIGH(x) String-handling

HIGH returns a character string of length x where each character is the highest character in the collating sequence (hexadecimal FF).

x expression specifying the length. If necessary, x is converted to a binary integer of precision (15,0).

INDEX(x1,x2) String-handling

INDEX returns a halfword binary integer indicating the starting position within the string x1 of a substring identical to string x2.

x2 string to be searched

x3 string to be searched for

If x2 does not occur in x2, the value zero is returned.

If x2 occurs more than once in x1, the starting position of the first occurence is returned.

If any argument is character or decimal, conversions are performed to produce character strings. Otherwise if the arguments are bit and binary, or both binary, conversions are performed to produce bit strings.

LENGTH(x) String-handling

LENGTH returns a default-precision fixed-point binary integer specifying the current length of a given string x. If x is binary, it is converted to bit string; otherwise conversions are performed to obtain a character string.

LOG(x)

The LOG built-in function returns a floarting-point value that is the base e (natural) logarithm of an arithmetic expression x. The computation is performed in floating point. The expression x must be greater than zero after its conversion to floating point.

MAX(x1,x2...,xn) Arithmetic

MAX returns, from a set of two or more arguments, the value of the argument with the largest value.

x1,x2...,xn list of values from which the largest is to be returned.

The maximum number of arguments that the function will accept is 64.

If the arguments are fixed-point with precisions:

(p1,q1), (p2,q2)...,(pn,qn)

the precision of the result is given by:

```
(MIN(N,MAX(p1-q1,p2-q2...,pn-qn)+
  MAX(q1,q2...,qn)),MAX(q1,q2...,qn))
```

If the arguments, after any necessary conversions have been performed, are floating point, and their precisions are p1,p2,p3...pn, then the precision of the result is MAX(p1,p2,p3...pn).

MIN(x1,x2...,xn) Arithmetic

MIN returns, from a set of two or more arguments, the value of the argument with the smallest value.

x1,x2...,xn list of values from which the smallest is to be returned.

The maximum number of arguments that the function will accept is 64.

If the arguments are fixed-point with precisions:

```
(MIN(N,MAX(p1-q1,p2-q2...,pn-qn)+
   MAX(q1,q2...,qn)),MAX(q1,q2...,qn))
```

If the arguments, after any necessary conversions have been performed, are floating point, and their precisions are p1,p2,p3...pn, then the precision of the result is MAX(p1,p2,p3...pn).

MOD(x1,x2) Arithmetic

MOD returns the smallest non-negative value, R, such that:

(x1-R)/x2 = n where n is an integer.

R is the smallest non-negative value that must be subtracted from a given value x1 to make it exactly divisible by the given value x2.

If x1 is positive, R is the remainder of the division of x1 and x2; if x1 is negative, R is the modular equivalent of this remainder.

If x2 is zero, the ZERODIVIDE condition is raised. If R is floating-point, the precision is the greater of those of x1 and x2; if R is fixed-point, the precision is given by:

(MIN(N,p2-q2+MAX(q1,q2)),MAX(q1,q2))

where (p1,q1) and (p2,q2) are the precisions of x1 and x2 respectively.

If x1 and x2 are fixed-point with different scale factors, R may be truncated on the right.

REPEAT(x1,x2) String-handling

REPEAT returns a string consisting of the string x1 concatenated to itself the number of times specified by x2, i.e. there will be (x2+1) occurrences of the string x1.

x1 string to be repeated

x2 an expression that can be converted to integer indicating number of repetitions.

If x1 is arithmetic, it will be converted to string - bit string if it is binary, character string if it is arithmetic. If x2 is zero or negative, the string x2 is returned.

If x2 is an array, then x1 should be an array with identical bounds.

ROUND(x1,x2) Arithmetic

ROUND returns the given value x1 rounded at a digit specified by x2.

x1 the value to be rounded.

x2 decimal integer constant, optionally signed, specifying the digit at which rounding is to occur. If x2 is positive, it is the (x2)th digit to the right of the point; if negative, it is the (x2+1)th digit to the left of the point.

If x1 is floating-point, x2 is ignored; the rightmost bit of the mantissa is set to 1.

The precision of a fixed-point result is given by:

(MAX(1,MIN(p-q+1+x2,N)),x2)

where (p,q) is the precision of x1 and N is the maximum number of digits allowable.

Note that the rounding of a negative value results in the rounding of its absolute value, then the sign is replaced.

SIGN(x) Arithmetic

SIGN returns a default-precision fixed-point binary integer that indicates whether a given value x is positive, zero, or negative. The value returned is as follows:

value of x	value returned
x > 0	+1
x = 0	0
x < 0	-1

SIN(x)

The SIN built-in function returns a floating-point value that is the sine of an arithmetic expression x, where x is an angle in radians. The sine is computed in floating point.

STRING(x) String-handling

STRING returns an element string that is the concatenation of all the elements of a string data aggregate.

x an array or structure expression whose elements are either all character strings and/or numeric character data, or all bit strings.

SUBSTR(x1,x2[,x3]) String-handling

SUBSTR returns a substring of the given string x1.

x1 string from which the substring is to be extracted.

x2 an expression that can be converted to integer specifying

the length of the substring in x1. If x3 is zero, a null string is returned. If x3 is omitted, the substring returned is position x2 in x1 to the end of x2.

If x1 is not a string, it is converted to a bit string if binary or a character string if decimal.

TIME

TIME returns a character string of length nine, in the form hhmmssttt, where:

hh	the current hour
mm	number of minutes
ss	number of seconds
ttt	number of milliseconds

If no timing facitlity is available, TIME returns the value (9)'0'.

TRANSLATE(x1,x2[,x3]) String-handling

TRANSLATE returns a string the same length as a given string x1 where all or some of the characters may have been changed. Characters are changed according to a look-up table provided by strings x2 and x3.

The function operates on each character of x1 as follows:

If a character in x1 is found in x3, then the character in x2 that corresponds to the one in x3 is copied to the result; otherwise, the character in x1 is copied directly to the result.

x1 character string to be searched for possible translation of all or some of its characters.

x2 character string containing the translation values of characters.

x3 character string containing the characters that are to be translated. If x3 is omitted, a string of 256 characters is assumed; it contains all possible characters arranged in ascending order (hexadecimal 00 through FF).

Strings x2 and x3 should be the same length; otherwise x2 is padded with blanks, or truncated, on the right to match the length of x3.

Any non-character arguments are converted to character.

UNSPEC(x) String-handling

UNSPEC returns a bit string that is the internal coded form of a given value x.

x expression of any data type

The length of the returned bit-string depends on the attributes of x.

If x is a varying-length string, its two-byte prefix is included in the returned bit-string.

bit-string length	attributes of x
16	FIXED BINARY(p,q) for p<16
32	FIXED BINARY(p,q) for p>15 FLOAT BINARY (p) for p<22

	FLOAT DECIMAL (p) for p>7 POINTER (standard length)
64	FLOAT BINARY (p) for 21<p<54 FLOAT DECIMAL (p) for 6<p<17
128	FLOAT BINARY(p) for 53<p<110 FLOAT DECIMAL(p) for 16<p<34
n	BIT (n)
n+16	BIT VARYING where n is the maximum length of x.
8*n	CHARACTER (n) PICTURE (with character-string length of n)
8*(n+2)	CHARACTER VARYING where n is the maximum length of x.
8*FLOOR(n)	FIXED DECIMAL (p,q) where n = (p+2)/2

The pseudovariable assigns a bit string directly to the given variable x, i.e., no conversion to the data type of the variable is attempted. The bit string is padded, if necessary, on the right with zeros to match the length of the variable. If x is a varying length string, its two-byte prefix is included in the field to which the bit string is assigned.

VERIFY(x1,x2) String-handling

VERIFY returns a default-precision fixed-point binary integer indicating the position in the given string x1 of the first character or bit that is not in the given string x2. If all the characters or bits in x1 do appear in x2, a value of zero is returned. The arguments are converted to strings if they are arithmetic. If one string argument is bit and the other character, the bit is converted to character.

x1 string to be scanned for any character not in x2.

x2 the verification string, consisting of a set of characters in any order.

If either argument is character or decimal, conversions are performed to produce character strings. Otherwise, if the arguments are bit and binary or both binary, conversions are performed to produce bit.

A.2 ADDED MODEL FUNCTIONS

AMAX(X1[,FOR_EACH.X2]*) System

Finds the maximum value of an array of elements. This function can be used across records. The optional FOR_EACH.X2 identifies the subscript on which the maximization is carried out. In the absence of any subscript the rightmost subscript of x1 is used.

AMIN(X1[,FOR_EACH.X2]*) System

Finds the minimum value of an array of values. This function can be used across records. The optional FOR_EACH.X2 identifies the subscript on which the minimization is carried out. In the absence of any subscript the rightmost subscript of x1 is used.

ANY(X1[,FOR_EACH.X2]*) System

This function denotes a selection of a single element out of an array of X1's which might be indexed (optionally) by the subscripts FOR_EACH_X2. In the absence of any subscripts the rightmost subscript of X1 is used.

RUNSUM(X1[,FOR_EACH_X2]*) Arithmetic

This function is identical in syntax and execution to that of SUM. The difference between them is that SUM accumulates the sum of the elements over the complete range of the subscripts implied while RUNSUM yields at any point the cummulative sum so far. Consider the two following examples:

I. For an input file containing a single field A in each record it is required to return a single record containing a field B which is the sum of all the fields A in the input file:

```
IN IS FILE(RECORD IS IN_R)(*));
 IN_R IS RECORD (A);
  A IS FIELD;
OUT IS FILE (RECORD IS OUT_R);
 OUT_R IS RECORD (B);
  B IS FIELD;
B = SUM (A);
```

The last assertion can also be written as:

```
B = SUM(A,FOR_EACH.IN_R);
B = SUM(A(FOR_EACH.IN_R)),
B = SUM(A(FOR_EACH.IN_R), FOR_EACH.IN_R);
```

II. consider now the case that for the same input file we wish an output file with an output record for each input record. The fileds in this record are C which is a copy of A, and D is the cumulative sum of all the A fields to this point.

```
IN IS FILE (RECORD IS IN_R)(*));
 IN_R IS RECORD (A);
  A IS FIELD;
OUT IS FILE (RECORD IS OUT_R)(*));
 OUT_R IS RECORD (C,D);
  C IS FIELD;
  D IS FIELD;
C = A;
D = RUNSUM(A)
```

The last assertion can also be written as:

```
D = RUNSUM(A, FOR_EACH.IN_R)
D (FOR_EACH.IN_R) = RUNSUM(A,FOR_EACH.IN_R)
```

etc.

SUM(X1[,FOR_EACH.X2]*) Arithmetic

X1 may be a variable or a subscripted variable. The X1 are summed. FOR_EACH.X2 is a subscript. In the absence of any FOR_EACH.X2 parameters the summation is performed on the rightmost subscript of X1. Note that in the presence of several subscripts as parameters a multiple level summation is performed.

REFERENCES

Cheng, T., Lock, E., & Prywes, N. S. (1983), "Use of program generation by accountants in the evolution of accounting systems: The case of financial reporting of changing prices," Paper presented at the Workshop on Reusability in Programming, ITT Programming, Rhode Island.

Lu, K. (1981), Program optimization based on a non-procedural specification, Contract N00014-76-C-0416, Information System Program, Office of Naval Research, Arlington, Va.

Lu, K. (1982), "MODEL program generator: System and programming documentation," Technical Report, Moore School of Electrical Engineering, University of Pennsylvania.

Prywes, N. S. (1978), MODEL II - Automatic program generator User Manual, Revision of Version 3, Contract TIR-77-41, Office of Planning and Research, Internal Revenue Service, Washington, D.C.

Prywes, N. S., & Pnueli, A. (1983), "Compilation of nonprocedural specifications into computer programs," IEEE Transactions on Software Engineering, Vol. SE-9, No. 3.

INDEX